UNIVERSIDADE FEDERAL DO PARANÁ

MATHEUS VINICIUS CORREA

TÉCNICAS DE PROJETO DE ALGORITMOS: BACKTRACKING E PROGRAMAÇÃO DINÂMICA

JANDAIA DO SUL
2017

MATHEUS VINICIUS CORREA

TÉCNICAS DE PROJETO DE ALGORITMOS: BACKTRACKING E PROGRAMAÇÃO DINÂMICA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção de título de Licenciado em Computação, Curso de Licenciatura em Computação, Universidade Federal do Paraná, *Campus* Avançado de Jandaia do Sul.

Orientador: Prof. Dr. Alexandre Prusch Züge

JANDAIA DO SUL
2017

Dados Internacionais de Catalogação na Publicação (CIP)

C824t Correa, Matheus Vinicius

Técnicas de projeto de algoritmos: Backtracking e programação dinâmica I. Matheus Vinicius Correa. Jandaia Do Sul, 2017.
90 f.

Orientador: Prof. Dr. Alexandre Prusch Züge.
Trabalho de Conclusão de Curso (Graduação – Licenciatura em Computação) – Campus Avançado da Universidade Federal do Paraná em Jandaia do Sul.

1. Algoritmos Combinatórios. 2. Problemas Combinatórios. 3. Programação Dinâmica. I. Züge, Alexandre Prusch. II. Universidade Federal do Paraná.

CDD 22. ed. 005.1

UNIVERSIDADE FEDERAL DO PARANÁ

PARECER Nº	**02 - AVALIAÇÃO MATHEUS VINICIUS CORREA/2017/UFPR/R/JA**
PROCESSO Nº	23075.217365/2017-50
INTERESSADO:	UFPR/R/JA/CCLC - COORDENAÇÃO DO CURSO DE LICENCIATURA EM COMPUTAÇÃO - JANDAIA

TERMO DE APROVAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO

Título: Técnicas de Projetos de Algoritmos: Backtracking e Programação Dinâmica

Autor: Matheus Vinicius Correa

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do grau no curso de Licenciatura em Ciência da Computação, aprovado pela seguinte banca examinadora.

- Prof. Dr. Alexandre Prusch Züge
- Prof. Me. Carlos Roberto Beleti Junior
- Prof. Dr. Marco Aurélio Reis dos Santos

Jandaia do Sul, 24/11/2017

Documento assinado eletronicamente por **ALEXANDRE PRUSCH ZUGE**, **VICE / SUPLENTE COORDENADOR DO CURSO DE LICENCIATURA EM COMPUTACAO**, em 01/12/2017, às 16:06, conforme art. 1º, III, "b", da Lei 11.419/2006.

Documento assinado eletronicamente por **CARLOS ROBERTO BELETI JUNIOR**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 04/12/2017, às 10:34, conforme art. 1º, III, "b", da Lei 11.419/2006.

Documento assinado eletronicamente por **MARCO AURELIO REIS DOS SANTOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 13/12/2017, às 10:10, conforme art. 1º, III, "b", da Lei 11.419/2006.

A autenticidade do documento pode ser conferida aqui informando o código verificador **0593036** e o código CRC **023E1B83**.

AGRADECIMENTOS

A Deus, pela força e fé alcançadas. Ao meu orientador Prof. Dr. Alexandre Prusch Züge, por ter sido muito mais que um orientador em muitos momentos, para o qual não tenho palavras para agradecer. Aos meus pais, Odinei e Raquel que sempre me apoiaram. Aos meus irmãos Muryllo e Mayara, que muito me incentivaram. A todos ótimos amigos que fiz e que me ajudaram em todos momentos. A todos aqueles que passaram em meu caminho e que de alguma forma me ajudaram até aqui.

RESUMO

O presente trabalho tem como objetivo apresentar técnicas de Projeto de Algoritmos, especialmente as técnicas de *Backtracking* e Programação Dinâmica, bem como estudos de caso da prática do ensino acerca destas técnicas. Para tanto, foi realizado um levantamento bibliográfico acerca das técnicas de Projeto de Algoritmos, focado na construção sólida dos principais conceitos inerentes a aplicação dessas técnicas. Foram escolhidos problemas computacionais NP-difíceis clássicos na literatura que pudessem apresentar à aplicação das técnicas citadas. Somados a estes, problemas oriundos de competições de programação são utilizados como prova de conceito. Os problemas computacionais foram escolhidos de acordo com características que apresentam em comum uns com os outros. Além disso, devido ao perfil do curso, foram selecionados casos e propostas de ensino das técnicas de *Backtracking* e Programação Dinâmica. Por fim, deve-se destacar que o estudo de técnicas de Projeto de Algoritmos ajuda os programadores no desenvolvimento de pensamento abstrato enquanto são abordados problemas de diversas áreas.

Palavras-chave: Problemas Combinatórios. Algoritmos Combinatórios. Memoização.

ABSTRACT

The present work aims to present Algorithm Design techniques, concentrating on Backtracking and Dynamic Programming algorithms, as well as to present case studies of the teaching practice about these techniques. For such end, a review on the literature was made on the techniques of Algorithm Design and the solid construction of the main concepts inherent to the application of these techniques. Classical NP-hard computational problems were chosen in the literature that could present the application of the mentioned techniques. In addition to these, problems from programming competitions are used as proof of concept. The computational problems were chosen according to characteristics they present in common with each other. In addition, due to of the course profile, cases and teaching proposals of the techniques of Backtracking and Dynamic Programming were selected. Finally, it is noteworthy that the study of Algorithm Design techniques helps programmers in developing abstract thinking while addressing problems of several areas.

Keywords: Combinatorial Problems. Combinatorial Algorithms. Memoization.

SUMÁRIO

1 INTRODUÇÃO

A Computação torna-se cada vez mais ubíqua, devido ao seu alto potencial na resolução de problemas, estando presente em campos como os da Física, Química, Medicina, Engenharia, Biologia e Economia. O domínio na arte de programação de computadores pode se fazer necessário nos mais variados contextos, uma vez que, em inúmeros campos do conhecimento é possível modelar um problema na forma de um *problema computacional*.

Um problema computacional é definido como um conjunto de instâncias na *entrada* que resulta em um conjunto de soluções na *saída*. Determinar se um número é primo é um exemplo de problema computacional. Determinar se o número 43 é primo — neste caso a saída é "sim" – é um exemplo de instância do problema. A solução de um problema computacional é um *algoritmo*: um método que determina uma sequência de passos finita para resolução de um problema. Algoritmos são um dos principais objetos de estudo da Ciência da Computação. Através de algoritmos eficientes é possível a resolução de uma grande variedade de problemas, incluindo alguns que seriam proibitivos se não fossem resolvidos com o auxílio de um computador.

Dependendo do problema a ser resolvido pode ser necessário uma combinação de conceitos matemáticos, lógicos e de Computação. Algoritmos é o campo no qual são estudados métodos eficientes de resolução de problemas. As técnicas de Projeto de Algoritmos é um conjunto de abordagens para a construção de algoritmos eficientes.

Dentre as técnicas de Projeto de Algoritmos podemos citar: Divisão e Conquista, Programação Dinâmica, Algoritmos Gulosos e *Backtracking*. Cada técnica resolve problemas que cabem em determinadas características e se distinguem nas limitações que apresentam. Nesse trabalho, serão abordadas as técnicas de Programação Dinâmica e *Backtracking*.

A programação dinâmica é uma técnica que busca encontrar uma solução ótima para problemas de otimização (máxima ou mínima). A técnica consiste na divisão do problema original em subproblemas menores dependentes entre si, resolve recursivamente cada problema e enfim, combina as soluções para resolver o problema original. Uma característica importante, é que durante seu procedimento, o algoritmo reduz o número de subproblemas salvando o valor de soluções já encontradas.

Backtracking é uma técnica que enumera todas soluções parciais de um dado problema. Essa é uma técnica que pode ser entendia como uma busca exaustiva que consiste em encontrar todas soluções que satisfazem restrições estabelecidas em determinadado problema. Problemas em que é preciso examinar todas soluções possíveis em busca de uma solução ótima, podem sugerir o uso de *algoritmos backtraking*.

Assim, este Trabalho de Conclusão de Curso aborda duas técnicas de Projeto de Algoritmos: *Backtracking* e Programação Dinâmica. Tais técnicas, assim como outras, são importantes do ponto de vista teórico e prático, possibilitando a resolução de um problema computacional em diferentes contextos. Exemplos de aplicações práticas passam por problemas

que emergem na indústria, ambientes acadêmicos e competições de programação ao redor do mundo. Considerando estes aspectos este trabalho visa possibilitar aporte a programadores, pesquisadores e demais interessados no assunto.

1.1 TÉCNICAS DE PROJETO DE ALGORITMOS E AS COMPETIÇÕES DE PROGRAMAÇÃO

Competições de programação são eventos que estimulam estudantes das áreas de algoritmos e estruturas de dados a buscarem soluções criativas para problemas computacionais em diferentes níveis de dificuldade. Um exemplo, é a Maratona de Programação[1], fase regional classificatória para a competição a nível mundial ACM-ICPC[2].

Em tais competições são recorrentes os problemas cuja solução passa pela aplicação adequada das técnicas de Projeto de Algoritmos. A identificação e estudo destas técnicas pode fazer com que participantes que desejem um bom desempenho direcionem seus estudos a esses assuntos, recorrendo a literatura ou a outras iniciativas que se propõem a fornecer material direcionado. É o caso do repositório *online* aberto ***Wikitona***[3], cujo o objetivo é disponibilizar material didático direcionado a competidores e interessados. O repositório foi criado como parte de um projeto de iniciação científica da Universidade Federal do Paraná e tem o objetivo de produzir materiais didáticos dos principais assuntos abordados em competições de programação, sobretudo a Maratona de Programação.

Além de repositórios *online*, existem livros direcionados à participantes de competições. O trabalho de Skiena e Revilla (2006) e o trabalho de Halim et al. (2013) são exemplos que se dedicam a citar técnicas como *Backtracking* e Programação Dinâmica como parte de uma preparação para competições.

1.2 OBJETIVO GERAL

O presente Trabalho de Conclusão de Curso tem como objetivo discorrer sobre o tema de Projeto de Algoritmos focando nas técnicas de *Backtracking* e Programação Dinâmica, atribuindo caráter prático às técnicas pela resolução de problemas exemplares.

1.3 OBJETIVOS ESPECÍFICOS

São objetivos específicos deste trabalho:

- Construir conhecimentos acerca das técnicas de Projeto de Algoritmos.
- Apresentar as técnicas de *Backtracking* e Programação Dinâmica.

[1] ver <http://maratona.ime.usp.br>
[2] ver <http://icpc.baylor.edu>
[3] ver <http://www.jandaiadosul.ufpr.br/wikitona>

- Resolver o problema da Oito Rainhas, da Clique Máxima e o Problema da Mochila utilizando a técnica *Backtracking*.
- Resolver os problemas da Mochila, Partição, Subsequência Mais Longa, Soma de Subconjuntos e Maior Subsequência Crescente por Programação Dinâmica.
- Relacionar o ensino de Algoritmos e as técnicas de Projeto de Algoritmos.
- Realizar prova de conceito com problemas oriundos da Maratona de Programação

1.4 JUSTIFICATIVA

O pensamento algorítmico aparece não só no campo da Ciência da Computação, mas também em outras áreas do conhecimento. Problemas computacionais compõem um dos objetos de estudos da Ciência da Computação, mas raramente são apresentados em termos matemáticos, em outras palavras, tendem a ser camuflados por detalhes de aplicação, alguns necessários, porém outros, não. Como resultado, o emprego de uma solução algorítmica consiste na identificação do núcleo matemático no problema e a identificação da técnica de projeto mais adequada. (KLEINBERG; TARDOS, 2006)

Ainda assim, muitos programadores profissionais não se sentem preparados para resolver problemas utilizando técnicas de Projeto de Algoritmos. Contudo, Projeto de Algoritmos faz parte do núcleo das principais *tecnologias* da Ciência da Computação. Projetar algoritmos para problemas do mundo real exige a compreensão de uma série de técnicas, incluindo estruturas de dados, Programação Dinâmica, busca em profundidade, *Backtracking*, entre outras. Um bom projeto se baseia no que se sabe sobre um dado problema e a partir de implementações já existentes tenta resolvê-lo. (SKIENA, 1998)

A técnica de programação dinâmica é uma técnica que tem como principal característica a diminuição do tempo de complexidade de um algoritmo. Essa técnica pode ser empregada na resolução do problema de encontrar a Maior Subsequência Crescente. O problema consiste em encontrar uma solução ótima que satisfaz a condição de que uma subsequência T de S, tem elementos estritamente crescente, na mesma ordem que estão em S, além disso, é a mais longa possível. Este problema é exemplar e conhecido por este autor no uso da técnica, mas não demonstra na totalidade o potencial da técnica. Desta forma, torna-se necessária o estudo de outros problemas que podem fornecer uma visão mais ampla do emprego da programação dinâmica.

Problemas combinatórios, como o problema de gerar todos os subconjuntos de um conjunto C, são exemplares na aplicação de algoritmos *backtracking*. Porém, existem outro problemas que exigem dos programadores maior poder de abstração na abordagem de resolução. Por exemplo, o problema das Oito Rainhas é um problema interessante que exemplifica o poder de abstração e o emprego de algoritmos *backtracking*.

O estudo aprofundado das técnicas presentes neste trabalho ajuda a aprimorar o pensamento abstrato dos programadores, quando os mesmos podem se deparar com problemas oriundos de diferentes áreas. De igual forma, corrobora na organização de uma solução viável e eficiente, visto que, por vezes uma solução de um problema não se trata apenas de aplicar as técnicas aqui reunidas, o que exige conhecimento prévio dos principais elementos que levam a solução mais adequada ao tipo de problema.

O pensamento computacional está presente na forma de como tarefas cotidianas são resolvidas. O desenvolvimento dessa habilidade ajuda a compreender e se relacionar com as tecnologias contemporâneas. As técnicas de Projeto de Algoritmos são exemplos mais aprofundados da aplicação do pensamento computacional na resolução de problemas.

1.5 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado em capítulos como se segue: o Capítulo 2 trata o aporte teórico apresentado na literatura. O Capítulo 3 e o Capítulo 4 tratam dos conceitos fundamentais presentes nesse trabalho. No Capítulo 5 são abordados algoritmos que resolvem problemas utilizando a técnica de *Backtracking*. No Capítulo 6 é apresentada a técnica de programação dinâmica e suas abordagens na resolução de problemas, considerando problemas exemplares do uso da técnica. Por fim, o Capítulo 7 apresenta as conclusões do trabalho.

Adicionalmente, implementações de diversos dos algoritmos abordados estão disponíveis no Apêndice A.

2 REVISÃO DA LITERATURA

A programação dinâmica é uma técnica que se aplica a problemas de otimização em que uma série de escolhas são realizadas, com o objetivo de alcançar uma solução ótima. Neste caso, tais escolhas são frequentemente relacionadas a subproblemas de mesma forma. A técnica é eficaz quando um conjunto de subproblemas emerge a partir de conjuntos potenciais de escolhas. Supondo que em um conjunto de escolhas, um mesmo subproblema aparece repetidas vezes, o artifício chave da programação dinâmica é armazenar a solução para cada subproblema já resolvido, resultado em um algoritmo mais eficiênte. (CORMEN, 2009).

Segundo Skiena e Revilla (2006), *Backtracking* é um método sistemático para iterar sobre todas possibilidades de configurações em um espaço de busca. É uma técnica geral, a qual, pode ser adequada a aplicações individuais. Segundo Kreher e Stinson (1999), um *algoritmo backtracking* é um método recursivo para construção viável de soluções para um problema combinatorial de otimização, um passo de cada vez. Um algoritmo backtracking é uma solução de busca exaustiva, isto é, todas soluções viáveis são consideradas e assim sempre encontrará a solução ótima.

O problema das Oito Rainhas, é o problema de posicionar oito rainhas em um tabuleiro de xadrez 8×8 de forma que um par qualquer de rainhas não se ataque. Contudo, não existe uma restrição se o problema for dispor n-rainhas em um tabuleiro $n \times n$. O problema das oito rainhas é um caso específico do problema das *n*-rainhas (SKIENA; REVILLA, 2006).

Dado um grafo $G = (V, E)$ onde V é um conjunto de vértices e E é um conjunto de arestas, uma clique é um conjunto $S \subseteq V$ que induz um grafo completo (BONDY; MURTY, 1976). O problema que procura pela clique de maior tamanho é o problema da Clique Máxima, um problema de otimização combinatório clássico com muitas aplicações. Algumas das muitas aplicações estão presentes no trabalho de Bomze et al. (1999). O problema da Clique Máxima é um problema provado $\mathcal{NP}$-difícil. (GAREY; JOHNSON, 2002)

Um problema clássico de programação dinâmica é o Problema da Mochila. Dasgupta, Papadimitriou e Vazirani (2006) descreve uma situação hipotética, onde, durante um assalto um ladrão encontra muito mais itens do que esperava e tem que decidir o que levar na pilhagem. Ele carrega uma mochila que suporta um peso total de W itens. Existem N itens para pegar, de peso $w_1, w_2, w_3, ..., w_n$ com valor $v_1, v_2, v_3, ..., v_n$ em lucro. Desta forma, pretende-se obter a combinação de itens mais valiosa que pode ser colocada na mochila. Segundo Sedgewick (1990), em programação dinâmica calculamos a melhor combinação para todas os tamanhos de mochila até o tamanho W para os i primeiros itens para um $0 \leq i \leq N$.

Resolver o problema da Maior Subsequência Comum é encontrar uma subsequência S de caracteres comuns a X e Y que seja a maior possível, dada as sequências X que tenha a forma $x_1, x_2, x_3, ..., x_n$ e uma sequência Y da forma $y_1, y_2, y_3, ..., y_n$. É possível resolver este problema por um algoritmo de força bruta, mas num tempo de complexidade $O(2^n m)$.

Felizmente uma abordagem por programação é mais eficiente. (GOODRICH; TAMASSIA; MOUNT, 2007).

O Problema das Moedas é discutido em Levitin e Mukherjee (2011), o qual é introduzido no seguinte contexto: é preciso retornar o troco em uma quantidade W usando a menor quantidade de moedas possíveis de um determinado tipo $v_1, v_2, \ldots, v_n$. Esse é um problema $\mathcal{NP}$-difícil, que pode ser visto como um Problema da Mochila sem limitante no número de moedas usadas na composição da resposta. (MARTELLO; TOTH, 1990)

O Problema da Partição é o problema $\mathcal{NP}$-completo no qual devemos decidir se é possível dividir um conjunto em dois subconjuntos, tal que, a soma dos elementos em cada subconjunto seja igual. O problema de dividir um conjunto em k subconjuntos de tamanho n é uma generalização desse problema (KORF, 1998).

Considerando uma sequência $S = s_1, s_2, \ldots, s_n$, o problema Maior Subsequência Crescente consiste em encontrar uma subsequência $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ na mesma ordem em que estão em S, onde, $1 \leq i_1 < i_2, \cdots, < i_k \leq n$, ou seja, uma subsequência crescente que atende a restrição de ser a maior possível (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006).

Em Andrei e Mahavier (2014) é discutida uma forma de ensino da estratégia de programação *Backtracking* através de jogos educacionais, em particular o jogo *Sudoku*. O ensino de programação dinâmica é abordada em Forišek (2015), onde o autor discute as abordagens encontradas na literatura e propõe uma forma de ensino. Ainda com relação ao ensino de programação dinâmica, ERDŐSNÉ NÉMETH e ZSAKÓ (2016) permeia a questão do ensino de algoritmos e estratégias de resolução de problemas. Por fim, Forišek e Steinová (2012) discute o ensino de algoritmos por metáforas.

Recurso educacional aberto é todo tipo de material de ensino, aprendizagem e investigação, digitais ou outros, que se situem no domínio público ou que tenha sido divulgado sob licença aberta que permite acesso, uso, adaptação e redistribuição gratuitos por terceiros (UNESCO, 2012). Dentro dessa perspectiva, a Wikitona é um repositório aberto em formato *wiki* direcionado a interessados e participantes de competições de programação (CORREA; ALBUQUERQUE; ZÜGE, 2016).

2.1 ENSINO

O impacto causado pela tecnologia da informação e comunicação nas relações da sociedade é um fato. A inserção da tecnologia no cotidiano demanda indivíduos com capacidade para usá-la da melhor maneira. Em consequência, a tecnologia exige competências que vão além do simples lidar com máquinas. (BRASIL, 2002)

Pensar de maneira abstrata é uma forma de lidar com a tecnologia disponível. Aqui, abstração é a definição de padrões, generalizando a partir de exemplos, distanciando seus detalhes concretos. Este tipo de pensamento, somado a outros, permite a utilização dos recursos disponíveis de maneira criativa. Pensar de maneira abstrata é um exercício mecânico, uma ferramenta, que permite a resolução de tarefas utilizando recursos computacionais, por exemplo.

Apesar de estar ligado a outros conceitos, como a Matemática, a aplicação de conceitos computacionais parece ser singular, sendo uma abordagem de pensar e resolver problemas. (BARCELOS; SILVEIRA, 2012)

O termo *pensamento computacional* foi apresentado pela comunidade acadêmica em um artigo publicado em março de 2006 (WING, 2006). Em tal artigo, o termo aparece como uma forma de articular uma visão ou modelo de pensamento que coubesse não apenas ao campo da Ciência da Computação, mas que pudesse ser abrangente o suficiente para caber em outras áreas. Por definição, *pensamento computacional* refere-se a habilidade de modelar e resolver tarefas aplicando conceitos computacionais, podendo ser essa solução executada por um humano, máquina ou pela combinação de humanos e máquinas.

A habilidade de empregar conceitos da Ciência da Computação na resolução de tarefas cotidianas pode estar relacionados a estratégias de Divisão e Conquista, Backtracking e Programação Dinâmica. Por exemplo, quando uma tarefa demanda muito esforço (cognitivo, coletivo, etc) é comum dividí-la em tarefa menores até que seja possível resolvê-la. Se um objeto está perdido, para encontrá-lo, uma estratégia é retroceder a lugares onde possivelmente ele esteja, fazendo lembrar *Backtracking*. Existem ideias[1] interessantes sobre como uma criança de quatro anos pode aprender o conceito de programação dinâmica.

Exemplo. Um exemplo de aplicação de pensamento computacional na resolução de problemas é a tarefa de um estudante que deseja carregar a menor quantidade de livros possível na mochila, sabendo quais as disciplinas ele tem em cada dia da semana. A Tabela 1 mostra a grade horária das aulas assistidas pelo estudante.

Tabela 1 – Distribuição das disciplinas nos dias da semana que o estudante tem aula.

Segunda-Feira	Terça-Feira	Quarta-feira	Quinta-feira	Sexta-feira
Matemática e Português	Português, Química e Artes	História e Matemática	História e Química	Matemática e Português

É possível iniciar analisando os dados do problema (livros, dias da semana, grade horária) para descobrir alguma relação ou padrão entre eles. Após, tentar criar uma sequência de passos que leva a uma resposta correta.

A resposta para esse exemplo passa por alguns dos conceitos relacionados ao pensamento computacional. Primeiro é realizada a coleta e análise dos dados presentes no problema. Em seguida uma sequência de passos que resolve a questão.

De fato o pensamento computacional incorpora outras habilidades que o definem. Os elementos a seguir fornecem bases para o ensino-aprendizagem do pensamento computacional, e estão de acordo com Grover e Pea (2013):

[1] ver <https://www.quora.com/How-should-I-explain-dynamic-programming-to-a-4-year-old>

- Abstração.
- Processamento de informação.
- Representação de informação.
- Notações de algoritmos.
- Interação, recursão e pensamento paralelo.
- Eficiência.
- Detecção de erros.

Existe um movimento para que tais habilidades sejam desenvolvidas através do ensino de Computação (GROVER; PEA, 2013). As técnicas de *Backtracking* e Programação Dinâmica podem ser vistas como técnicas onde o pensamento computacional e seus elementos são aplicados de maneira aprofundada em resolução de problemas. Sendo assim, a utilização dessas técnicas como ferramenta para o desenvolvimento das habilidades de pensamento computacional pode ser notada. A Seção 2.1.1 aborda o ensino de *Backtracking* e Programação Dinâmica.

2.1.1 Propostas

A Universidade Federal do Paraná oferece, em seu câmpus avançado localizado na cidade de Jandaia do Sul, um curso de Licenciatura em Computação que objetiva formar profissionais habilitados a trabalhar como docentes, bem como, implementar, aprimorar e utilizar ferramentas aplicáveis aos vários espaços e níveis educacionais (PPC, 2015). Dito isso, torna-se relevante contribuir com ensino das técnicas Backtracking e Programação Dinâmica pelo estudo de alguns casos.

Em Andrei e Mahavier (2014) é discutido uma forma de ensino da estratégia de programação *Backtracking* através de jogos educacionais, em particular o jogo *Sudoku*. Apesar de ser conhecido com outros nomes, Sudoku é mencionado em 1986 por uma companhia japonesa de jogos. Seu nome pode ser traduzido do japonês como "os dígitos devem ser únicos". O objetivo é preencher uma grade 9×9 com números de 1 a 9. A grade é dividida em setores menores 3×3. Os números devem ser colocados de maneira tal que não seja repetidos na mesma coluna, linha e setor. O jogo inicia-se parcialmente preenchido com valores e a dificuldade está em preencher o restante com números de 1 a 9 satisfazendo a restrição dada. A Figura 1 apresenta duas grades: (a) representa um jogo inicial; (b) a resposta para o jogo.

Em geral esse é um problema que apresenta uma única solução. A proposta de ensino discutida em Andrei e Mahavier (2014) é de estudar a solução para o Sudoku utilizando o pseudocódigo de um algoritmo geral de Backtracking. A ideia consiste em partir disso e gerar a arvore de estados e examinar cada estado. O pseudocódigo foi construído para levar à uma

Figura 1 – Exemplo de jogo Sudoku. (a) é configuração inicial dada e (b) é resposta para jogo. Os números em vermelho em (b) foram acrescentados durante a resolução.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Jogo inicial

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(b) Solução para o jogo

árvore de estados que revela uma única solução. Algumas modelagens matemáticas também são realizadas com o intuito de auxiliar na obtenção de uma resposta.

O ensino de programação dinâmica é abordada em Forišek (2015), onde o autor discute as abordagens encontradas na literatura e propõe uma forma de ensino. Em livros-textos, principalmente, o caminho para apresentação da técnica é pela utilização de exemplos resolvidos por programação dinâmica. Segundo o autor as abordagens presentes na literatura costumam falhar em:

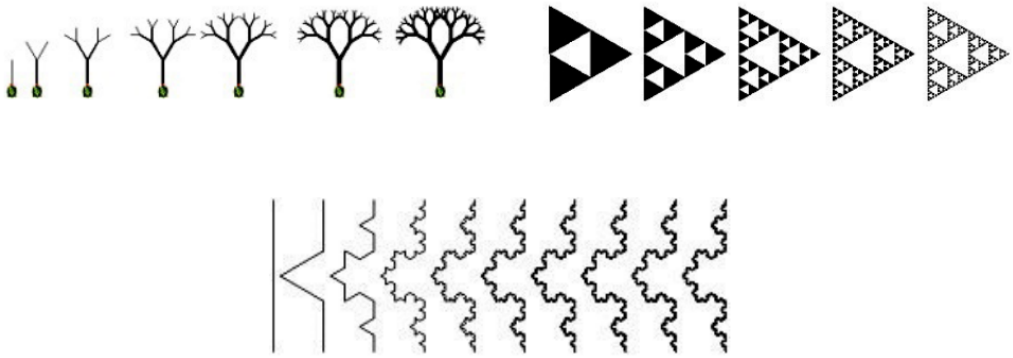
1. Falta de motivos lógicos para escolher e ordenar tarefas.
2. Avisos sobre potenciais armadilhas de solução de algum problema.

Para atacar essas fraquezas o autor sugere uma abordagem de resolução de problemas por programação dinâmica seguindo os passo:

1. Implementar algoritmos recursivos.
2. Usar o algoritmo para descobrir uma relação de recorrência entre todas soluções possíveis.
3. Adicionar memoização para melhorar seu tempo de complexidade.
4. Converter a solução em uma solução iterativa *bottom-up*.

Ainda com relação ao ensino de programação dinâmica, ERDŐSNÉ NÉMETH e ZSAKÓ (2016) permeia a questão do ensino de algoritmos e estratégias de resolução de problemas. É discutida de forma sucinta os diferentes tipos de programação dinâmica e como eles podem ser ensinados desde muito cedo, na infância, até a fase adulta. Nos primeiros anos do estudo de matemática e informática a construção de funções recursivas podem ser deixadas de lado, oferecendo um recurso visual como a Figura 2 para introduzir o assunto. A autora

Figura 2 – Exemplos de árvores de recursão



acredita que o ensino pode ser feito gradativamente durante todo período escolar do aluno, prosseguindo com problemas cada vez mais difíceis exigindo uma reflexão cada vez maior dos conceitos de expressões recursivas, otimização e memoização.

Por fim, Forišek e Steinová (2012) discute o ensino de algoritmos por metáforas. Metáforas e analogias são úteis em qualquer nível de educação. No contexto da Ciência da Computação exemplos de analogias como a de "etiquetas" para uma variável de um programa são frequentes. Assim, os autores discutem analogias possíveis para diversos conceitos da Computação, como pilhas, recursão e Geometria Computacional.

Existem outras iniciativas no ensino de Algoritmos. A Wikitona é um ambiente aberto online onde são apresentados diversos algoritmos. O ambiente é voltado a participantes de competições de programação que buscam aprimorar suas habilidades como programadores. Competições de programação costumam cobrar dos candidatos um domínio amplo de várias habilidades e técnicas de resolução de problemas. Assim, a Wikitona tem como objetivo abordar a maior quantidade possível de conteúdos relacionados à competições (CORREA; ALBUQUERQUE; ZÜGE, 2016). Dentro da perspectiva de Recurso Educacional Aberto (UNESCO, 2012), a Wikitona é um repositório aberto em formato *wiki*. Contudo, ainda não existe um conteúdo voltado as técnicas *Backtracking* e Programação Dinâmica, fazendo desse, um ambiente em potencial para trabalhos futuros.

3 PRELIMINARES

Esta seção visa definir conceitos e notações importantes para a apresentação deste trabalho. Os conceitos fundamentais em Combinatória é parte importante para os estudos de problemas combinatórios. A resolução algorítmica de problemas combinatórios são uma das motivações para o estudo de algoritmos *Backtracking* e Programação Dinâmica. Algoritmos *Backtracking* podem ser usados para gerar todas permutações de uma dada sequência ou gerar todas combinações de subconjuntos existentes em um conjunto de objetos. Em Programação Dinâmica estamos interessados em algoritmos que levam a uma solução de combinações ótima (máxima ou mínima).

No Problema da Mochila é preciso encontrar a combinação de itens com peso associado e valor associado que podem ser colocados em uma mochila durante um saque. A escolha dos itens deve ser a feita de forma a maximizar o lucro obtido com o saque sem ultrapassar a capacidade máxima da mochila. O problema apresenta em si uma característica de problema combinatório, uma vez que, é preciso encontrar a combinação ótima que respeite as restrições do problema.

Outro problema que exemplifica o uso adequado de conceitos combinatórios é o problema das Oito Rainhas. O problema das Oito Rainhas, é um problema no qual o conhecimento prévio dos conceitos de permutação cria uma solução algorítmica adequada ao problema.

Combinatória é o ramo da Matemática que trata da contagem. Tratar contagem é importante sempre que temos recursos finitos ou sempre que estamos interessados em eficiência. Problemas de contagem se resumem em determinar quantos elementos existem um conjunto finito. Apesar de parecer uma tarefa fácil, a contagem pode não ser (GERSTING, 2004).

As seções seguintes nesse capítulo estão organizadas como se segue: As Seções 3.1 e 3.2 são as bases para as demais seções e, a Seção 3.3 apresenta definições de conceitos e notações utilizadas neste trabalho. Em seguida a Seção 3.3.1 define os problemas discutidos entre os demais capítulos. A Seção 3.4 define tipos estruturas usadas neste trabalho. As Seções 3.5, 3.5.1 e 3.6 tratam da definição, solução e classificação de problemas combinatórios.

3.1 PERMUTAÇÕES

Uma permutação é um arranjo de n itens, onde cada item aparece exatamente uma vez. Existe $n! := \prod_{i=1}^{n} i$ diferentes permutações. Em geral o número de permutações de r objetos escolhidos dentre n objetos distintos é denotado por $P(n, r)$. (GERSTING, 2004)

O valor de $P(n, r)$ é dado por

$$P(n, r) = \frac{n!}{(n-r)!} \text{ para } 0 \leq r \leq n$$

Exemplo. O valor de $P(7,3)$ é

$$\frac{7!}{(7-3)!} = \frac{7!}{4!} = \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{4 \cdot 3 \cdot 2 \cdot 1} = 7 \cdot 6 \cdot 5 = 210$$

Exemplo. O número de permutações de três objetos a, b e c, é dado por $P(3,3) = 3! = 3 \cdot 2 \cdot 1 = 6$. São elas,

$$abc, acb, bac, bca, cab, cba$$

3.2 COMBINAÇÕES

Combinação refere-se ao ato de selecionar k objetos de um conjunto de n objetos, de maneira tal que (em contraste com a permutação) a ordem não importa. Em outras palavras, estamos escolhendo k objetos distintos dentre n possíveis e contando o número de combinações. O número de subconjuntos com k elementos sobre um conjunto com n elementos é igual ao **coeficiente binomial**, denotado $\binom{n}{k}$ (lê-se n escolhe k), cujo valor é

$$\binom{n}{k} = \frac{n!}{k!(k-n)!} \text{ para } 0 \leq k \leq n.$$

Exemplo. O valor de $\binom{7}{3}$ é

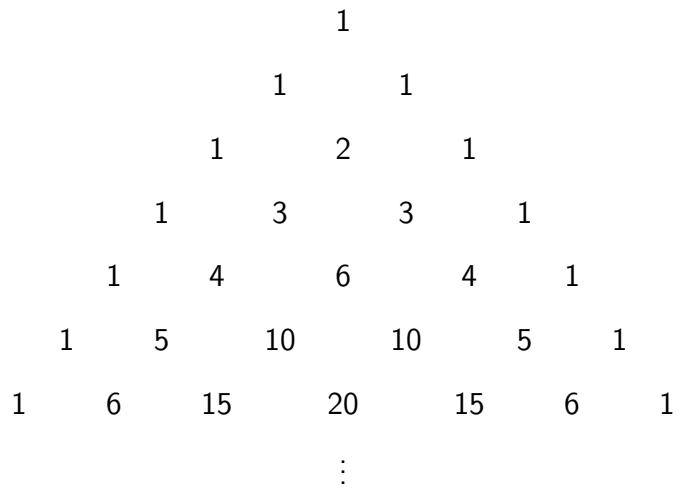
$$\frac{7!}{3!(7-3)!} = \frac{7!}{3!4!} = \frac{7 \cdot 6 \cdot 5}{3 \cdot 2 \cdot 1} = 7 \cdot 5 = 35$$

Exemplo. Com quantas maneiras podemos formar um comitê com 4 membros dentre 9 pessoas? Aplicando a definição de coeficiente binomial, temos $\binom{9}{4} = 126$ comitês.

Usando o conceito de coeficiente binomial, é possível determinar o k-ésimo elemento da n-ésima linha do *Triângulo de Pascal*. Arranjando os coeficientes binomiais, cuja primeira linha é $n = 0$, o triângulo tem a forma

$$\begin{array}{lccccccccc}
n=0: & & & & & \binom{0}{0} & & & & \\
n=1: & & & & \binom{1}{0} & & \binom{1}{1} & & & \\
n=2: & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & \\
n=3: & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & \\
n=4: & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} \\
\vdots & & & & & \vdots & & & &
\end{array}$$

Calculando os valores para as expressões anteriores, obtêm-se os valores do triângulo apresentados da seguinte forma



Para algum n e k não negativo, podemos observar que o elemento $\binom{n}{k}$ do triângulo é obtido pela soma dos dois elementos nas diagonais logo acima. A *Fórmula de Pascal* dá uma forma de gerar o triângulo por

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{3.1}$$

A interpretação da equação 3.1 é que desejamos calcular o número de maneira que podemos escolher k objetos dentre n. Supondo que exista um elemento particular x de um conjunto com n elementos. Sempre que k elementos são escolhidos, existem duas decisões possíveis para o elemento x: x pertence ao subconjunto escolhido ou não. Ora, se x é um dos k elementos então restam $k-1$ elementos a serem escolhidos dentre $n-1$, logo $\binom{n-1}{k-1}$ maneiras. No caso em que x não é escolhido, é preciso tomar k elementos dentre $n-1$ elementos diferentes de x, para o que existem $\binom{n-1}{k}$ maneiras. Pela soma desses dois eventos disjuntos, temos o total de maneira que podemos escolher k objetos dentre n.

Outra propriedade do triângulo de Pascal é que a soma dos elementos de cada linha é uma potência de dois, especificamente

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

.

3.3 DEFINIÇÕES E NOTAÇÕES

É importante ter em mente os tipos e as formas como algumas estruturas são representadas. É possível definir estruturas para várias finalidades, tais como, agrupar e representar elementos ou objetos comuns em uma determinada característica. São utilizados os seguintes termos e notações no decorrer deste trabalho.

Conjuntos – Um conjunto é uma coleção de objetos distintos entre si, chamados de *elementos* ou *membros*. Os elementos de um conjunto podem ser números, letras, outros conjuntos,

grafos, etc. O conteúdo de um conjunto é escrito entre chaves. Por exemplo, os conjuntos seguintes:

$A = \{\text{João}, \text{Maria}, \text{Pedro}\}$	conjunto de nomes comuns.
$B = \{a, b, c\}$	conjunto das primeiras letras do alfabeto.

Esta notação cabe aos conjuntos *finitos*. Conjuntos *infinitos* podem ser descritos como

$$C = \{2, 3, 5, 7, 11, \dots\} \text{ conjunto dos números primos.}$$

Um conjunto não leva em consideração a ordem em que os elementos aparecem, desta forma, $\{7, 8, 9\} = \{7, 9, 8\} = \{9, 8, 7\}$. Não utilizamos uma notação para multiconjuntos. O conjunto $\{a, a\}$ indica que o elemento a está presente no conjunto, por isso, $\{a, a\} = \{a\}$. A notação $s \in S$ indica que s é um elemento do conjunto S, lida como "s pertence a S". E $s \notin S$ indica que s não pertence a S.

O *tamanho* ou *cardinalidade* de um conjunto S, denotado por $|S|$, é o número de elementos presentes em um conjunto. Por exemplo, $|\{6, 7, 8, 9\}| = 4$. O conjunto vazio é um conjunto que não contém nenhum elemento, o qual denotamos $\emptyset$. Assim $|\emptyset| = 0$. Se o tamanho de um conjunto é determinado por um número inteiro positivo, então esse conjunto é **finito**; caso contrário, o conjunto é **infinito**.

Considerando dois conjuntos X e Y, a notação $X \subseteq Y$ no diz que X é subconjunto de Y. Se X é subconjunto de Y, então todo elemento de X é um elemento de Y; caso contrário, $X \nsubseteq Y$.

O *produto cartesiano* entre dois conjuntos S e T, denotado $S \times T$, é o conjunto dos pares ordenados entre os elementos S e T, tal que, o primeiro elemento do par ordenado é um elemento de S e o segundo elemento pertence a T. Portanto,

$$S \times T = \{(s, t) : s \in S \text{ e } t \in T\}.$$

Por exemplo,

$$\{1, 5, 6\} \times \{0, 1, 2\} = \{(1, 0), (1, 1), (1, 2), (5, 0), (5, 1), (5, 2), (6, 0), (6, 1), (6, 2)\}.$$

Sequências – Uma sequência é uma coleção de objetos ordenados, os quais são chamados de **membros** ou **termos**. Tomando a sequência $S = (1, 0, 5, 4, 8, 1)$, descrevemos os seus termos com a notação de subscritos, tal como $s_1, s_2, \dots, s_n$, cujo termo $s_1 = 1$ é o primeiro termo, $s_2 = 0$ o segundo termo e assim por diante até o n-ésimo termo da sequência. Sequências são denotadas entre parênteses, como (a, b, c). Quando o sentido estiver claro pelo contexto, os parênteses podem ser omitidos. Sequências e conjuntos exercem a mesma função no que se refere a agrupar uma coleção de objetos, mas se distinguem em vários aspectos.

Conjuntos devem ter seus itens distintos entre si, enquanto que em sequências, seus membros podem ser iguais. Assim, $A = (1, 3, 1)$ é uma sequência de tamanho três, mas $B = \{1, 3, 1\}$ é um conjunto com dois elementos.

Como já mencionado, os termos de uma sequência levam em consideração a ordem que aparecem. As sequências (a, b, c) e (c, b, a) são distintas, contudo $\{a, b, c\} = \{c, b, a\}$.

O tamanho de uma sequência é o número de seus termos. Por exemplo, $S = (1, 3, 4, 1)$ é uma sequência de tamanho 4. Uma sequência com um número inteiro positivo n de termos é chamada sequência finita, a qual frequentemente é chamada de **n-upla**. Uma 2-upla também é chamado **par**. Se existem 0 elementos em uma sequência, essa sequência é vazia, frequentemente denotada por **( )** ou por λ. A sequência $(1, 4, 8, \dots)$ é uma sequência infinita. Uma sequência finita de letras é chamada de **string**.

A Sequência de Fibonacci é um tipo de sequência que recebe uma notação especifica F_n, que indica calcular o n-ésimo termo da sequência. Existem outras sequências que também recebem uma notação particular, como C_n – números de Catalan.

Grafo – Seguindo a notação presente em Bondy e Murty (1976) um grafo é um par ordenado $G = (V(G), E(G))$ formado por um conjunto de *vértices* $V(G)$ e um conjunto de *arestas* $E(G)$ que tem relação entre si. Um grafo **direcionado** tem o conjunto de arestas $E(G)$ formado por pares ordenados (u,v). Em um grafo **não direcionado** o conjunto de arestas $E(G)$ consiste de pares *não ordenado* $\{u, v\}$, onde $u, v \in V(G)$.

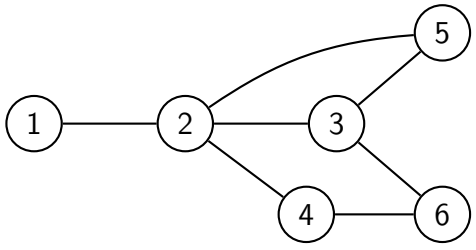
Um grafo comumente é representado de forma visual. Os elementos do conjunto de vértices são desenhados como círculos, podendo ser identificadas ou não. As arestas que relacionam dois vértices u e v são representadas por uma linha que os conecta. A Figura 3 representa um grafo G onde o conjunto $V(G) = \{1, 2, 3, 4, 5, 6\}$ e $E(G) = \{(1, 2), (2, 3), (2, 4), (3, 5), (3, 6), (4, 6), (5, 2)\}$.

Um grafo H é dito *subgrafo* de G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Se existir um conjunto $S \subseteq V$, o subgrafo $G[S] = \left(S, E(G) \cap \binom{S}{2}\right)$ é o *subgrafo induzido* pelo conjunto S; dizemos que $G[S]$ é um subgrafo induzido de G.

Um grafo G é *completo* se todo par distinto de vértices é ligado por uma aresta, ou seja, G é completo se $E(G) = \binom{V(G)}{2}$.

Durante este trabalho, será frequente a referência a um grafo G, não sendo necessário a distinção entre grafos por diferentes letras. Neste caso, adotamos V e E ao invés de $V(G)$ e $E(G)$ omitindo a letra G.

Figura 3 – Representação visual de um grafo $G = (V(G), E(G))$



3.3.1 Definições de Problemas Computacionais

Serão uteis definições de problemas computacionais discutidos com frequência no decorrer do presente trabalho. Um problema computacional é um problema que pode ser formulado em termos de uma *instância* na **entrada** e uma **saída**. Definimos os seguintes problemas computacionais.

Problema das Rainhas

O Problema das Rainhas é um problema combinatório, onde a resposta passa pela permutação no número de rainhas presentes em um tabuleiro de xadrez. Enunciamos o problemas das n rainhas como se segue.

Problema das Rainhas

Em um tabuleiro de xadrez de dimensões $n \times n$ e n rainhas de um jogo comum de xadrez. É possível dispor todas as n rainhas no tabuleiro de forma que cada rainha não ataque nenhuma outra?

O Problema das Rainhas é exemplar na aplicação da técnica de construção de algoritmos *Backtracking*. O problema das Oito Rainhas, é um caso especial do problema das n rainhas onde $n = 8$. Uma discussão mais aprofundada está presente na Seção 5.2. O problema computacional das n rainhas pode ser definido como se segue.

Problema das Rainhas	
Instância:	um inteiro n que representa as dimensões de um tabuleiro e a quantidade de rainhas.
Devolva:	"sim" se é possível dispor n rainhas em um tabuleiro de xadrez $n \times n$; "não", caso contrário.

Problema da Mochila

O Problema da Mochila é um problema de otimização com muitas aplicações. Ele fornece um modelo que podemos aplicar a outros problemas relacionados a investimentos de capital, cortes, empacotamento, carregamento de veículos e orçamentos (KELLERER; PFERSCHY; PISINGER, 2003). Na criptografia, foi um modelo utilizado em chaves públicas, como o sistema de criptografia Merkle-Hellman (SHAMIR, 1984). Enunciamos o problema da mochila em seguida.

Problema da Mochila

Um ladrão está realizando um saque e deseja encher uma mochila de capacidade M com itens de diferentes pesos $w_1, w_2, \ldots, w_n$ e lucros $v_1, v_2, \ldots, v_n$, de maneira que a capacidade W não seja ultrapassada e o lucro obtido pelo saque seja o maior possível.

O problema computacional relacionado ao Problema da Mochila leva em consideração que os lucros e pesos são vetores na instância de entrada. A capacidade W pode ser representada como um inteiro. Portanto, durante este trabalho é considerado o Problema da Mochila booleana (0-1 *Knapsack*). Definimos o problema em seguida.

Problema da Mochila	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W.
Devolva:	Melhor lucro possível que não ultrapasse a capacidade W.

É possível resolver o problema utilizando algoritmos *backtracking* e programação dinâmica, apresentados nas Seções 5.5 e 6.2.1. O Problema da Mochila pode servir como o modelo no qual outros problemas podem ser interpretados. Decidir se existe um subconjunto de A que somado obtém um valor W, é o problema da *Soma de Subconjunto* discutido na seção 6.2.1, que pode ser interpretado como um Problema da Mochila. (KELLERER; PFERSCHY; PISINGER, 2003)

Problema das Moedas

Frequentemente, nos deparamos com a situação em que um atendente deve nos retornar um troco por alguma compra, desejando contar o troco com o menor número de moedas possível. Enunciamos em seguida o Problema das Moedas.

Problema das Moedas

Um caixa tem um estoque ilimitado de moedas com valores $v_1, v_2, \ldots, v_n$. Dado um inteiro W, qual o menor número de moedas que somadas obtém W?

Este problema é exemplar, pois sua resolução pode ser modelada pela aplicação direta da técnica de programação dinâmica. Além disso, pode ser interpretado em termos do Problema da Mochila. Este problema é discutido com mais detalhes na Seção 6.2.3. O problema computacional é definido formalmente como se segue.

Problema das Moedas	
Instância:	*moedas* $v_1, v_2, \ldots, v_n$;
	objetivo W.
Devolva:	Número mínimo de moedas que somadas obtém W.

Problema da Partição

Considerando um conjunto de inteiros positivos $A = \{9, 4, 5\}$, o problema da partição é o problema de decidir se existe dois subconjunto A_1 e A_2 tal que a soma dos elementos em A_1 seja igual a soma dos elementos de A_2. Para esse caso, somando os elementos do conjunto $A_1 = \{9\}$ e $A_2 = \{4, 5\}$ é possível verificar que a restrição do problema é atendida. Dizemos que esse problema é o problema 2-partição, um caso particular do problema da k-partição, para um k inteiro. Enunciamos o problema em seguida.

Problema da Partição

Seja o conjunto A um conjunto de inteiros positivos e k um inteiro positivo, decidir se existem subconjuntos $A_1, A_2, \ldots, A_k$ de tamanho n cujo a soma dos elementos em A_1 seja igual a soma dos elementos em $A_2, A_3, \ldots, A_k$.

O Problema da k-partição, assim como o Problema da Mochila e o Problema as Moedas é um problema que pode ser resolvido pela técnica de programação dinâmica. A seção 6.2.2 aborda uma solução algorítmica. O seguinte problema computacional é equivalente ao Problema da k-partição.

Problema da K-Partição	
Instância:	*conjunto* T;
	inteiro k.
Devolva:	"sim" se é possível dividir T em k subconjuntos de tamanho n;
	"não", caso contrário.

Problema da Subsequência Comum Mais Longa

O problema da Subsequência Comum Mais Longa é o problema de encontrar a subsequência de tamanho máximo que seja comum as sequências X e Y. Enunciamos o problema em seguida.

Subsequência Comum Mais Longa

Dado duas cadeias $X = x_1, x_2, \ldots, x_n$ e $Y = y_1, y_2, \ldots, y_n$ encontrar uma subsequência $Z = z_1, z_2, \ldots, z_n$ comum ao mesmo tempo a X e Y com o maior tamanho possível.

Computacionalmente o problema é definido como se segue.

Subsequência Comum Mais Longa	
Instância:	sequências X e Y.
Devolva:	subsequência comum mais longa.

Dado as sequências $X = A, B, C, G, B, A$ e $Y = F, G, B, A, C, B, A$, a sequência B, C, A é comum a X e Y. A sequência B, C, A não é a subsequência comum mais longa, pois tem tamanho três. A subsequência B, C, B, A é comum a X e a Y e tem o maior tamanho possível.

Problema da Maior Subsequência Crescente

O problema da Maior Subsequência Crescente é um problema estudado em ramos da Matemática e Física. O problema é enunciado como se segue.

Maior Subsequência Crescente

Dado uma sequência de inteiros S encontrar a maior subsequência em ordem crescente contida em S.

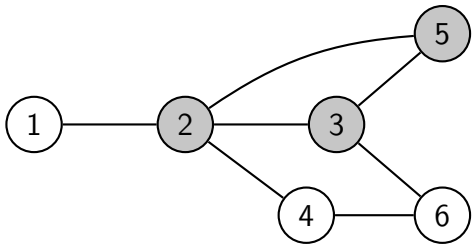
Seja a sequência $S = (7, 3, 5, 2, 7, 8, 1)$: uma subsequência contém os elementos $A_1 = (5, 3, 1)$; a sequência $A_2 = (3, 5, 8)$ é uma subsequência crescente; a maior subsequência cresente é a que contém os elementos $A_3 = (3, 5, 7, 8)$. O problema computacional é definido formalmente como se segue.

Maior Subsequência Crescente	
Instância:	sequência S.
Devolva:	maior subsequência crescente.

3.4 ESTRUTURAS COMBINATÓRIAS

As estruturas combinatórias são as que podem ser descritas como coleções de k-elementos, k-uplas, ou permutações de um conjunto. Dado esse tipo de estrutura estamos

Figura 4 – Clique máxima em um grafo G



interessados em investigar todas subestruturas presentes dentro de um tipo particular. (KREHER; STINSON, 1999).

Problema da Clique Máxima

No problema da Clique Máxima o objetivo é encontrar uma determinada estrutura característica de um grafo. Este é um problema combinatório de interesse teórico e prático com muitas aplicações citadas em Bomze et al. (1999).

Uma *clique* em um grafo G é um subconjunto $S \subseteq V$ onde $G[S]$ é completo. O problema da Clique Máxima é o problema que procura pela clique de maior tamanho em um grafo G. A Figura 4 apresenta uma clique de tamanho três, composta pelos conjunto de vértices $S = \{2, 3, 5\}$. Essa é a única e maior clique do grafo.

Utilizando a notação descrita neste capítulo, descrevemos um problema computacional equivalente a encontrar a clique máxima. A Seção 5.3 aborda a solução do problema da clique máxima.

Problema da Clique Máxima (CM)	
Instância:	Um grafo G.
Devolva:	Uma clique máxima de G.

3.5 PROBLEMAS COMBINATÓRIOS

Problemas combinatórios procuram por grupos, ordem ou características de objetos discretos finitos que satisfazem uma condição dada. Resolver problemas utilizando um ou vários princípios de Contagem sugere à problemas combinatórios.

Estudar os princípios de problemas combinatórios facilita no desenvolvimento, análise e apresentação de algoritmos. Algoritmos combinatórios, são algoritmos que resolvem problemas combinatórios ou problemas de estrutura combinatória. As seções subsequentes descrevem algoritmos, estruturas e problemas combinatórios.

3.5.1 Tipos de Problemas Combinatórios

Existem vários tipos de problemas combinatórios, caracterizados pelo tipo de resposta que se espera (KREHER; STINSON, 1999). Os exemplos seguintes utilizam o Problema da Mochila para demonstrar tal ideia.

Um *problema de decisão* é um problema modelado como uma **pergunta** ou **questão** a ser respondida com "sim" ou "não". É esperado para esse tipo de problema, respostas do tipo "verdadeiro" ou "falso", "possível" ou "não possível", "existe" ou "não existe".

O Problema da Mochila tem um problema de decisão associado. Pode ser escrito considerando as condições do problema original, porém é adicionada uma condição (mínima) que deve ser satisfeita. Definindo um valor P é desejada uma solução no mínimo igual a esse valor em lucro. Nessa modelagem, a resposta do problema é sinalizar se a condição foi alcançada.

Problema da Mochila (Decisão)	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W;
	lucro objetivo P
Questão:	existe sequência $x_1, x_2, \ldots, x_n \in \{0,1\}^n$, tal que,
	$\sum_{i=1}^{n} v_i x_i \geq P$
	e
	$\sum_{i=1}^{n} w_i x_i \leq W$?

O Problema da Mochila (Decisão) é um problema em que a resposta esperada é sim ou não. Algoritmos que resolvem este tipo de problema devem responder corretamente (sim ou não) para qualquer instância. (KREHER; STINSON, 1999).

Por vezes, é preciso encontrar uma solução de maneira exata. O Problema da Mochila pode ser estendido para o problema que de fato encontra a sequência que satisfaça todas condições. Definimos essa versão do problema em termos de um problema que busca contruir a resposta e devolvê-la.

Problema da Mochila (Busca)	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W;
	lucro objetivo P
Questão:	encontrar sequência $x_1, x_2, \ldots, x_n \in \{0,1\}^n$, tal que,
	$\sum_{i=1}^{n} v_i x_i \geq P$ e $\sum_{i=1}^{n} w_i x_i \leq W$ se tal sequência existe.

O Problema da Mochila (Busca) corresponde ao problema de decisão, mas é solicitado a sequencia $x_1, x_2, \ldots, x_n$ no caso em que a resposta à busca é "sim". (KREHER; STINSON, 1999).

A versão de encontrar um valor ótimo do Problema da Mochila, não estabelece um valor de lucro para ser alcançado. O objetivo é alcançar o maior valor possível de um lucro P que satisfaça as restrições do problema (KREHER; STINSON, 1999). O valor ótimo pode ser calculado para **máximo** ou **mínimo**.

Problema da Mochila (valor ótimo)	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W;
Questão:	o valor máximo de $P = \sum_{i=1}^{n} v_i x_i$
	sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$
	e $x_1, x_2, \ldots, x_n \in \{0,1\}^n$.

No Problema da Mochila (Otimização) é preciso encontrar a sequência $x_1, x_2, \ldots, x_n$ que satisfaça as condições já definidas cujo lucro seja ótimo. (KREHER; STINSON, 1999)

Problema da Mochila (Otimização)	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W;
Questão:	uma sequencia $x_1, x_2, \ldots, x_n \in \{0,1\}^n$, tal que,
	$P = \sum_{i=1}^{n} v_i x_i$ seja máximo,
	sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$.

Em problemas de otimização existem várias restrições a serem satisfeitas. As sequências de $x_1, x_2, \ldots, x_n$ que satisfazem as restrições estabelecidas são chamadas de **solução viável** ou **solução possível**.

3.6 ALGORITMOS COMBINATÓRIOS

São algoritmos combinatórios, os algoritmos que investigam estruturas combinatórias classificados informalmente de acordo com seu propósito. Segundo Kreher e Stinson (1999) as classificações podem ser feita em *algoritmos de geração*, *enumeração* e *busca*.

Geração – *Constrói* todas estruturas combinatórias de um tipo. Estruturas de exemplo são subconjuntos, permutações, partições, árvores.

Enumeração – *Calcula* o número de estruturas diferentes de um mesmo tipo. Todo algoritmo de geração é um algoritmo de enumeração quando cada objeto possa ser contado enquanto é gerado. O número de k-subconjuntos de n-elementos é $\binom{n}{k}$. Contudo, não procedemos da mesma forma em estruturas que podem assumir diferentes representações. Esta ideia está presente em estruturas *isomorfas*.

Busca – Um problema típico de busca é de encontrar uma clique de um tamanho específico em um grafo. Algoritmos de geração podem ser usados para encontrar uma estrutura em particular, mas em muitos problemas não será eficiente.

Uma variação do problema do problema de busca, são os **problemas de otimização**, onde queremos encontrar a estrutura ótima de um dado tipo. Uma estrutura ótima é definida de acordo a algum critério de "lucro" ou "prejuízo".

4 ANÁLISE DE ALGORITMOS

Quando um algoritmo é projetado, torna-se importante saber fazer certas inferências sobre ele. Por exemplo, se está sempre correto para qualquer instância de um dado problema, ou ainda, quanto tempo ele demora para produzir um resultado. Tais inferências caminham para uma análise dos *recursos* que uma implementação consome, podendo ser analisados em relação ao tempo de execução e espaço de memória consumidos.

Tais recursos podem ser afetados por diferentes implementações e poder de processamento de um *hardware* que executa um algoritmo. Contudo, a análise de algoritmos não leva em consideração aspectos externos aos passos que um algoritmo faz para chegar a uma resposta. De fato, poder generalizar uma analise trás muitos benefícios. Assim, são utilizados métodos matemáticos para dizer quanto recurso um algoritmo vai tomar. Esses recursos podem ser mensurados utilizando medidas de **tempo de execução**, também chamado de **complexidade de tempo**, que indica quão rápido um algoritmo realmente é em relação ao tamanho da entrada de uma instância.

Algoritmos podem ser analisados independentemente da linguagem ou poder de processamento de uma máquina que o executa. Na verdade não é preciso implementá-los em linguagem de programação. As principais técnicas que possibilitam análise da eficiência dos mais variados algoritmos, passa pelo estabelecimento de um modelo computacional e análise assintótica de pior caso. (SKIENA, 1998)

Para realizar a análise de um algoritmo independentemente de recursos de máquina é preciso considerar um modelo de computação, como o modelo hipotético RAM (*Máquina de acesso aleatório*, em inglês), ver Aho e Hopcroft (1974). O modelo RAM fornece um modelo de computação simplificado muito aproximado a de um computador *real*. A medida que um algoritmo é executado sobre esse modelo, será possível contar a quantidade de passos realizados para resolver uma dada instância com o intuito de generalizar essas medidas.

Contar a quantidade de passos que um algoritmo realiza é um método que auxilia na quantificação do tempo de execução dada uma instância de entrada. Ao empregar esse método em dado algoritmo, talvez seja notável que ao passo que o tamanho da entrada muda (aumenta, diminui, dobra, etc.) o tempo de execução também muda. Com isso, é possível estabelecer uma função em relação ao tamanho da entrada n. Essa função pode ser uma função modelada como qualquer função numérica já conhecida, com $f(n) = n^2 + 4n + 2$.

Neste ponto a entrada de um algoritmo é um número n que pode indicar o número de elementos de uma lista, dimensões de uma matriz, etc. A magnitude desse número determina o tamanho de uma entrada. Em algoritmos que resolvem problemas como os de fatoração em números primos ou teste de primalidade, o tamanho da entrada é determinada pela magnitude de n. Em Levitin e Mukherjee (2011) para casos como esses, o tamanho da entrada pode ser

melhor entendido como o número b de *bits* usados na representação binária de n, dado por

$$b = \lfloor \log_2 n \rfloor + 1.$$

4.1 COMPLEXIDADE DE TEMPO

Com auxilio do modelo de computação RAM, é possível determinar o tempo de execução de um algoritmo sobre uma instância com entrada de tamanho n. Para entender o comportamento de um algoritmo em tempo de execução, é preciso ter em mente que existem instâncias que levam mais tempo que outras, de modo que, será necessário saber seu comportamento para todas as instâncias.

É possível estabelecer funções de complexidade de tempo de acordo com as instâncias que alimentam um algoritmo: *pior caso*, *melhor caso* e *caso médio*. As definições de complexidade de tempo segundo Skiena (1998) são:

- A *complexidade de pior caso* de um algoritmo é uma função definida pelo número máximo de passos que ele executa sobre qualquer instância de tamanho n.
- *Complexidade de melhor caso* é uma função definida pelo menor número de passos que um algoritmo faz em todas instâncias de tamanho n.
- *Complexidade de caso médio* de um algoritmo é uma função definida pelo número médio de passos tomados sobre todas instâncias de tamanho n.

O estudo da complexidade de tempo de um algoritmo geralmente é concentrada na análise de pior caso. Com um trabalho moderado, essa abordagem permite fazer afirmações sobre a eficiência de um algoritmo – dada pela função definida pelo número máximo de passos que ele executa em relação ao tamanho da entrada. Assim, por essa abordagem, obtêm-se um *limite superior* de tempo de execução de um algoritmo.

Sendo um limite superior, o pior caso não acontece em todas entradas, uma vez que em muitas entradas o tempo de execução de fato será muito menor. Isso pode levar a uma análise mais intuitiva no caso médio. Analisar um algoritmo levando em conta a complexidade de caso médio prova-se como uma tarefa difícil do ponto de vista matemático.

4.2 NOTAÇÃO ASSINTÓTICA

Analisar um algoritmo com foco na complexidade de pior caso pode ser uma tarefa mais fácil em relação ao caso médio. Ainda assim, modelar uma função de pior caso em termos do tamanho de uma entrada n como a uma função $3n^3 + 2n + 1$ pode não ser uma tarefa fácil. Ao invés disso, uma tarefa mais simples, pode ser utilizar limites inferiores ou superiores como n^3, omitindo detalhes que não impactam a comparação entre funções. Desta forma, uma analise se torna focada nos fatores que realmente impactam uma função que descreve o tempo

de execução de um algoritmo – sua taxa de crescimento. Em Knuth (1998) os conceitos de notação assintótica e taxa de crescimento de funções são discutidos com mais detalhes, onde são inauguradas as discussões acerca da análise de algoritmos utilizando notações assintóticas.

Considerando este trabalho, na maioria das vezes em que é aplicada a notação assintótica a uma função, esta notação tem a interpretação de descrição da complexidade de tempo. Contudo, essa notação pode ser interpretada como outras características dependendo do contexto em que se apresenta (quantidade de espaço, por exemplo).

Com a finalidade de comparar e poder fazer certas afirmações sobre taxas de crescimento, são usadas algumas notações: O (Oh grande), Ω (omega grande) e Θ (teta). Estas notações seguem as definições em Horowitz e Sahni (1978), as quais são descritas como:

- **Notação** O: Supondo $f : \mathbb{N} \to \mathbb{R}$ e $g : \mathbb{N} \to \mathbb{R}$. Uma função $f(n) = O(g(n))$ (lê-se f de n é *oh* grande de g de n) se e somente se existir constantes positivas c e n_0, tais que, $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.

  A afirmação $f(n) = O(g(n))$ mostra que $g(n)$ é um **limite superior** de valores de $f(n)$ para todo $n, n \geq 0$. Ela não diz nada a respeito de quão bom é esse limitante. Para ser uma informação representativa, o limitante superior deve ser idealmente a menor função que se pode encontrar para $f(n) = O(g(n))$ ser verdadeiro. Dessa forma, é adotado $2n + 1 = O(n)$ e não $2n + 1 = O(n^2)$, embora este último não esteja errado.

  Notações frequentes: a notação $O(1)$, diz que seu tempo de execução é constante. A notação $O(n)$ indica tempo linear, $O(n^2)$ é chamada de quadrática. Denotamos $O(2^n)$ uma função de tempo exponencial no pior caso.

- **Notação** Ω: Supondo $f : \mathbb{N} \to \mathbb{R}$ e $g : \mathbb{N} \to \mathbb{R}$. Uma função $f(n) = \Omega(g(n))$ (lê-se f de n é omega de g de n) se o somente se existir constante positiva c e n_0, tal que, $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$.

  Uma função $g(n)$ é um **limite inferior** de $f(n)$, ou seja, a afirmação $f(n) = \Omega(g(n))$ diz que existe constante c, tal que, $f(n)$ é sempre $\geq cg(n)$, para n suficientemente grande. Para ser significativo, o limitante inferior deve ser a maior função que se pode encontrar para $f(n) = \Omega(g(n))$ ser verdadeira. Assim, as adotamos $2n + 1 = \Omega(n)$ e $4^n + n^2 = \Omega(4^n)$ ao invés de $2n + 1 = \Omega(1)$ e $4^n + n^2 = \Omega(n)$, embora ambas as afirmações sejam válidas.

- **Notação** Θ: Supondo $f : \mathbb{N} \to \mathbb{R}$ e $g : \mathbb{N} \to \mathbb{R}$. Uma função $f(n) = \Theta(g(n))$ (lê-se f de n é teta de g de n) se e somente ser existir constantes positivas c_1, c_2 e n_0, tal que, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$.

  A definição de Θ tenta ser a que tem a noção mais precisa entre O e Ω. A afirmação $f(n) = \Theta(g(n))$ é verdadeira se e somente se $g(n)$ é um limite superior e um limite inferior sobre $f(n)$. Esta definição apresenta um limitante justo sobre $f(n)$. A função

$2n + 1 = \Theta(n)$ pois $2n + 1 \geq 2n$ para todo $n \geq 1$ e $2n + 1 \leq 3n$ para todo $n \geq 2$, assim $c_1 = 2$ e $c_2 = 3$, com $n_0 = 2$.

No presente trabalho a principal notação utilizada será a notação $O(\cdot)$, que é interpretada como o comportamento de um algoritmo no pior caso, uma vez que fornece um limite superior para uma dada função.

4.3 ORDENS DE CRESCIMENTO

Existem funções de tempo que aparecem com frequência em vários algoritmos. Essa frequência é notada quando aplica-se a ideia de ignorar fatores constantes na função, adotando a notação assintótica. A Tabela 2 mostra a taxa de crescimento de cinco funções frequentemente vistas na análise de complexidade de algoritmos. Conforme o valor da entrada n cresce é possível observar a variação de tempo em segundos que um algoritmo pode demorar para resolver um dado problema.

Tabela 2 – Taxas de crescimento de funções sobre um computador que executa um milhão de instruções por segundo.

	Função de n				
Valor de n	n	$n \log_2 n$	n^2	2^n	$n!$
10	<1 s	<1 s	<1 s	1 s	4 s
50	<1 s	<1 s	<1 s	36 anos	–
100	<1 s	<1 s	<1 s	10^7 anos	–
1000	<1 s	<1 s	1 s	–	–
10 000	<1 s	<1 s	2 min	–	–
1 000 000	1 s	20 s	12 dias	–	–

Com base em observações como as da Tabela 2, citando Kleinberg e Tardos (2006), uma definição empírica de algoritmo eficiente é um algoritmo que tem tempo de execução polinomial, ou seja, $O(n^k)$ para um inteiro $k > 0$.

De fato não é preciso analisar o tempo de execução de um algoritmo executando-o em uma máquina e contando o tempo que leva para resolver uma instância de um problema. A notação O é uma ferramenta que possibilita essa análise considerando apenas os termos de maior grau de uma função a medida que o n cresce.

A análise de algoritmos tem outra forma de abordagem como em (SEDGEWICK; FLAJOLET, 1996) que foca diretamente na análise de tempo, sendo mais precisa.

5 BACKTRACKING

Computadores modernos estão cada vez mais rápidos, a ponto que algoritmos de força bruta podem ser escolhas eficientes na resolução de problemas. Por exemplo, às vezes torna-se mais fácil contar o número de itens em um conjunto de objetos se efetivamente o construir utilizando algoritmos combinatórios. Naturalmente, existem limites no número de itens que podem ser gerados em tempo aceitável. (SKIENA; REVILLA, 2006)

É possível considerar que computadores pessoais modernos tem uma taxa de *clock* maior que 1 GHz, o que significa que são capazes de realizar mais de um bilhão de operações por segundo, ou seja, existe a possibilidade de buscar por alguns milhões de itens por segundo em máquinas atuais.

Para uma melhor noção do que significa a quantidade de *milhões*, basta computar os arranjos de 10 ou 11 objetos que implica em algo em torno de um milhão de permutações. Outra proporcionalidade se dá ao gerar um milhão de subconjuntos, o que significa todas combinações de 20 itens grosseiramente. Problemas com entradas maiores, requererem estratégias um pouco mais sofisticadas no trato de tantas operações, de maneira que seja construído algoritmos eficientes. (SKIENA; REVILLA, 2006).

Um *algoritmo de backtracking* é um método recursivo para construir todas as soluções viáveis sobre um problema combinatório. Um algoritmo de *backtracking* é um algoritmo de busca exaustiva que sistematicamente gera todas as possíveis soluções e escolhe a melhor. É possível encontrar um conjunto de soluções parciais que não leva a solução ótima. Neste caso a busca deve ser interrompida. *Poda* é processo que pode ser usado para interromper a busca por uma solução que nunca será ótima. (KREHER; STINSON, 1999)

5.1 ALGORITMO DE BACKTRACKING GERAL

Para muitos problemas combinatórios é possível modelar uma solução como um vetor $a = (a_1, a_2, \ldots, a_n)$ onde cada elemento a_i é selecionado de um conjunto de possibilidades P_i. Os elementos do vetor a são gerados um de cada vez, em ordem. Esse tipo de vetor pode representar um arranjo onde a_i contém o i-ésimo elemento de uma permutação. O vetor a pode representar um conjunto $C \subseteq P$, onde a_i é verdadeiro se e somente se o i-ésimo elemento está em C. O vetor a também pode representar uma sequência de movimentos em um jogo de tabuleiro ou um caminho de um grafo, onde a_i contém o i-ésimo evento na sequência. (SKIENA; REVILLA, 2006)

De forma genérica, um algoritmo de *backtracking* tenta expandir uma *solução parcial* $a = (a_1, a_2, \ldots, a_k)$, adicionando outro elemento ao final. Para todo elemento a_k de uma solução parcial existe um subconjunto $C_k \subseteq P_k$ que chamamos de **conjunto de candidatos**. C_k é computado com base em regras que restringem cada elemento $c \in C_k$. Toda configuração

do vetor a que satisfaz uma restrição é uma solução viável. Ao encontrar uma solução viável o algoritmo *processa* essa solução.

O algoritmo Backtrack(k) implementa uma versão geral dessa ideia. Esse algoritmo serve como um modelo que pode ser especializado de acordo com o problema – definindo as regras para computar C_k e processar uma solução possível. O vetor a é definido como sendo *global* e o conjunto C_k é gerado a cada chamada recursiva.

```
Backtrack(k)
    Entrada: inteiro k; vetor global C_k; vetor global a.
    Saída: vetor a.
1   Se a é uma solução possível então
2   |   processa o vetor a
3   Senão
4   |   C_k ← candidatos a expandir a
5   |   Para c ∈ C_k faça
6   |   |   a_k ← c
7   |   |   Backtrack(k + 1)
```

Backtracking garante a solução correta uma vez que enumera todas soluções possíveis.

A mesma ideia apresentada no algoritmo Backtrack(k) está presente no algoritmo de Busca em Profundidade. Por isso, o algoritmo Backtrack(k) pode ser visto também, como uma implementação de busca em profundidade sobre um grafo implícito. (SKIENA; REVILLA, 2006)

Exemplo. Construir todos os subconjuntos de um conjunto de n itens. Considerando o algoritmo Backtrack(k) como um modelo, o algoritmo Subconjuntos(k) é um algoritmo que enumera todas subconjuntos de um conjunto. Serão gerados todos 2^n subconjuntos de n itens. Isso equivale a gerar 2^n vetores com elementos 0 ou 1 onde 0 significa que o item i não está no subconjunto e 1 que o item i é um elemento do subconjunto. A solução parcial $a = (a_1, \ldots, a_k)$ é viável quando $k \geq n$.

```
Subconjuntos(k)
    Entrada: inteiro k; vetor global C_k; vetor global a.
    Saída: vetor a.
1   Se k = n então
2   |   processa o vetor a
3   Senão
4   |   C_k ← {0, 1}
5   |   Para c ∈ C_k faça
6   |   |   a_k ← c
7   |   |   Subconjuntos(k + 1)
```

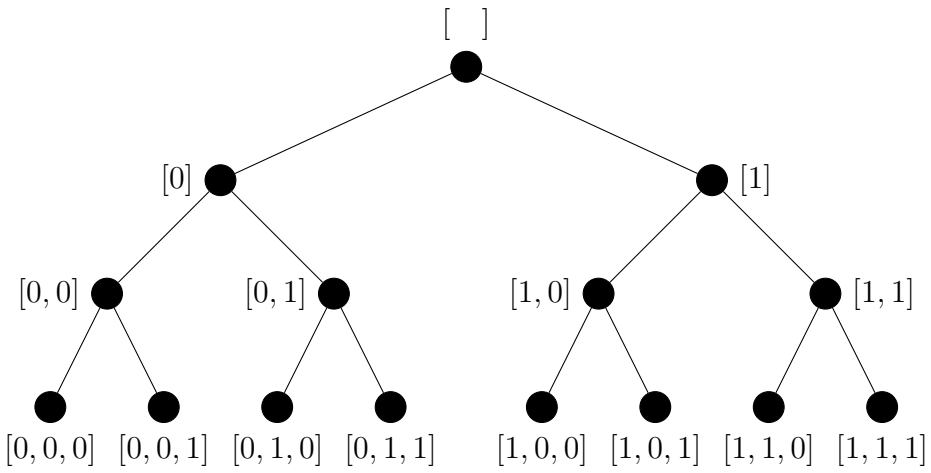
Este é um algoritmo ingênuo para construir todas 2^n possíveis n-uplas na ordem em que uma busca em profundidade é realizada. Analisando o tempo de execução do algoritmo,

Subconjuntos(k) toma $\Theta(1)$ para checar se uma solução é viável. É possível afirmar que o algoritmo leva a um tempo de execução proporcional a $\Theta(2^n)$, desconsiderando o tempo necessário para processar o vetor. Conforme n cresce, esse algoritmo pode se tornar cada vez mais impraticável.

Exemplo. Um exemplo prático utilizando o algoritmo Subconjuntos(k) é gerar todos subconjuntos de $\{1, 2, 3\}$. Quando a é uma solução viável, *processar o vetor* a equivale a imprimir os valores para os quais $a_i \neq 0$. Neste caso, o algoritmo Subconjuntos(k) imprime todos os subconjuntos: $\emptyset, \{3\}, \{2\}, \{2, 3\}, \{1\}, \{1, 3\}, \{1, 2\}, \{1, 2, 3\}$.

A Figura 5 apresenta uma árvore binária onde cada nó é uma chamada recursiva do algoritmo Subconjuntos(k) quando $n = 3$. Este tipo de árvore leva o nome de **árvore de estados** (*state tree*, em inglês), a qual não necessariamente tem a forma de uma árvore binária. Cada nó de uma árvore de estados define um estado. Todos os caminhos a partir do nó raiz formam um *espaço de estados*. Uma *solução possível* é o caminho da raiz até um estado s. Os nós internos de uma árvore de estados podem ser interpretados como soluções parciais de uma instância. Devido a estas características de uma árvore de estados e semelhança com a busca em profundidade, a ideia de um **espaço de busca** também é usada para se referir ao conjunto de estados gerados por um algoritmo. (KREHER; STINSON, 1999; HOROWITZ; SAHNI, 1978)

Figura 5 – Árvore de estados do algoritmo Subconjuntos(k) quando $n = 3$.



5.2 PROBLEMA DAS OITO RAINHAS

O problema das Oito Rainhas é um problema exemplar no uso do algoritmo *Backtracking*. Este problema é um caso particular do problema das n rainhas enunciado como se segue.

Problema das Rainhas

Seja um tabuleiro de xadrez de dimensões $n \times n$. Além disso, n rainhas de um jogo comum de xadrez. É possível dispor todas as n rainhas no tabuleiro de forma que uma rainha não ataque a outra?

Dados n rainhas e um tabuleiro de xadrez de um jogo comum com dimensões $n \times n$, é preciso decidir se é possível dispor as n rainhas de forma que nenhuma rainha consiga atacar outra. O problema de dispor **oito** rainhas nos mesmos critérios pode ser visto como um caso especial do problema das n-rainhas. Assim, tomando um tabuleiro de dimensão 8×8 e os movimentos que as 8 rainhas podem exercer durante um jogo de xadrez, é preciso decidir se é possível dispor as 8 rainhas em um tabuleiro de dimensão 8×8 de maneira, tal que, não exista um ataque entre duas rainhas.

Retomando da seção 3.3.1 onde definimos o problema computacional das rainhas em que o problema é definido como se segue.

Problema das Rainhas	
Instância:	um inteiro n que representa as dimensões de um tabuleiro e a quantidade de rainhas.
Devolva:	"sim" se é possível dispor n rainhas em um tabuleiro de xadrez $n \times n$; "não", caso contrário.

Uma resposta para este problema pode ser obtida pelo emprego de uma estratégia de busca sobre todas possibilidades de dispor 8 rainhas no tabuleiro. Essa abordagem pode ser implementada sobre o algoritmo Backtrack(k). Para implementar algoritmos *backtracking* é preciso escolher com sabedoria a maneira de representar uma solução viável na forma de um vetor, visando a forma mais eficiente.

Uma primeira ideia de representação considera todos 2^{n^2} subconjuntos de um tabuleiro formados por 0 ou 1. Utilizando a ideia de enumeração de subconjuntos, nessa representação o elemento a_i será *verdadeiro* se a i-ésima casa do tabuleiro não é atacada por nenhuma outra rainha e *falso* caso contrário. As casas do tabuleiro são enumeras de 1 a n^2 representadas por um vetor $a = (a_1, \ldots, a_{n^2})$. Uma solução possível é gerada após todas n^2 casas conterem valores onde n delas são valores iguais a verdadeiro. Para $n = 8$ existem $2^{64} \approx 1{,}84 \times 10^{19}$ vetores.

Outra ideia corresponde em representar uma solução na qual o elemento a_i é um valor entre 1 e n^2, ou seja, uma lista que diz a casa em que a i-ésima rainha está alocada. Uma solução é dada quando os primeiros n itens de a são preenchidos adequadamente. Os candidatos a i-ésima posição de a são as casas que não são atadas pelas primeiras $i - 1$ rainhas.

Isso é proporcional a $(n^2)^8$ possíveis vetores, o que é equivalente a $64^8 \approx 2{,}81 \times 10^{14}$ vetores quando $n = 8$.

As ideias anteriores para resolver o problema das rainhas são ineficientes. Para tornar um algoritmo backtracking eficiente é preciso eliminar as soluções parciais que não levam a uma solução ótima, ou seja, *podar* a maior parte das possibilidades antes de efetivamente construir cada uma delas.

5.2.1 Poda

Poda é uma processo que tenta diminuir o espaço de busca interrompendo a expansão de uma solução parcial que não pode levar a uma solução viável para um problema (SKIENA; REVILLA, 2006). Problemas combinatórios tendem a ter um crescimento muito acentuado em tempo de execução. Enumerar todos subconjuntos de n itens leva tempo exponencial, uma permutação entre itens de uma lista leva $O(n!)$. Para que um algoritmo backtracking seja rápido o suficiente para resolver problemas interessantes é preciso podar os ramos da árvore que não levam a uma solução viável ou possível.

No problema das Oito Rainhas, uma solução que enumera todos subconjuntos é impraticável. Sendo assim, surge a pergunta: É possível fazer melhor? Para não existir ataques, duas rainhas não podem estar na mesma linha do tabuleiro. É possível numerar as rainhas de 1 a 8, assim como as linhas de 1 a 8. Dessa forma, a rainha i está na linha i de mesmo valor. As soluções para o problema podem ser representadas como 8-uplas $(a_1, \ldots, a_8)$, onde a_i é uma coluna na qual a rainha i está disposta. Assim, existem oito colunas candidatas em $C_k = \{1, 2, 3, 4, 5, 6, 7, 8\}$ para cada peça sobre uma linha, reduzindo o espaço de busca para o equivalente $8^8 = 16.777.216$ tuplas.

É possível melhorar ainda mais. A ideia da solução anterior permite repetições nas colunas. Em uma análise mais empírica, duas rainhas não podem ocupar a mesma coluna, linha ou diagonal. Estas restrições implicam que a_i não pode ser repetido. O vetor a passa a ser uma permutação entre as 8 rainhas, reduzindo de 8^8 tuplas para $8! = 40320$ possibilidades de resposta viável.

Baseado no modelo de construção de um algoritmo backtracking, é possível implementar uma solução que procura pela permutação que satisfaça as restrições do problema. Durante a construção do algoritmo, deve existir um cuidado no passo que define os candidatos para a posição a_i, assegurando que as análises anteriores estão sendo atendidas.

No algoritmo Rainhas(k) uma posição segura nos indica uma casa do tabuleiro onde é possível posicionar uma rainha de maneira que não ocorra um ataque. Podemos verificar uma posição segura percorrendo um vetor a. Se uma posição é segura ela faz parte do conjunto C_k de candidatos que podem ocupar a i-ésima posição da permutação. O Apêndice A.1 disponibiliza uma implementação do algoritmo Rainhas(k) em linguagem de programação C++.

O algoritmo devolve um valor maior que 0 se é possível dispor n rainhas em um tabuleiro $n \times n$. Definimos sua a chamada inicial com $k = 0$. Para o problema das Oito

Rainhas(k)

Entrada: inteiro k; vetor global C_k; vetor global a.
Saída: inteiro represetando o número de soluções possíveis.

```
1 Se k = n então
2 |   incrementa o número de soluções possíveis
3 Senão
4 |   k ← k + 1
5 |   C_k ← k-ésima posição segura
6 |   Para c ∈ C_k faça
7 |   |   a_k ← c
8 |   |   Rainhas(k)
```

Rainhas existem 92 soluções viáveis. Considerando que exista uma função apropriada para gerar os elementos de C_k, garantindo que a solução seja uma permutação entre n elementos, o algoritmo leva tempo $O(n!)$ para chegar em todas soluções possíveis.

5.3 PROBLEMA DA CLIQUE MÁXIMA

Outro problema combinatório interessante é o problema da Clique Máxima (CM). Como apresentado na seção 3.4, uma clique Q em um grafo $G = (V(G), E(G))$ é um subconjunto de $V(G)$ tal que $G[Q]$ é completo. O *número de clique* de G frequentemente denotado $\omega(G)$ é o tamanho da clique máxima definido por $\omega(G) := max\{|Q| : Q \text{ é uma clique de } G\}$. O problema da CM é o problema de encontrar uma clique Q em grafo G de maior tamanho possível. Este é um problema $\mathcal{NP}$-difícil e sua versão de decisão é um dos primeiros problemas demonstrados como sendo $\mathcal{NP}$-completo citado na lista de 21 problemas $\mathcal{NP}$-completos de Karp (1972).

Existe uma relação entre uma clique máxima e um conjunto independente. O conjunto de vértices $S \subseteq V(G)$ é chamado de *conjunto independente* se $E(G[S]) = \emptyset$. Notamos que uma clique S em G é um conjunto independente no grafo complementar $\overline{G}$ tal que $\overline{G} := \left(V(G), \binom{V(G)}{2} - E(G)\right)$.

O problema da CM é um problema de otimização conhecido para o qual existe uma literatura bem consolidada. Este é um problema de muitas aplicações, várias descritas em Bomze et al. (1999). Uma aplicação interessante surge em sistemas distribuídos interligados por Internet. Em certos casos para a realização de um teste é necessário escolher um conjunto de computadores com a rede mais estável possível. Encontrar conjuntos estáveis pode ser feito pela busca de uma clique máxima em um grafo formado por uma rede de computadores. A solução para este problema é abordada em Duarte et al. (2010).

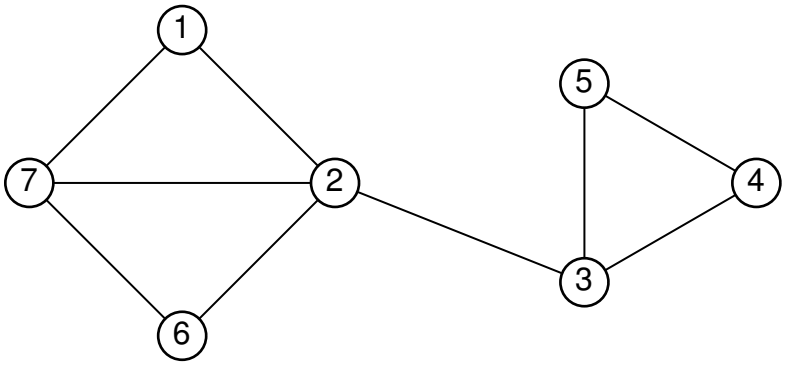
Utilizando as notações anteriormente apresentadas, enunciamos o problema da CM da seguinte forma.

Problema da Clique Máxima

Dado um grafo $G = (V, E)$, o problema da Clique Máxima busca pela clique de maior tamanho em G, ou seja, o conjunto de vértices que induz um grafo completo onde $\omega(G)$ seja um valor máximo.

Exemplo. A Figura 6 apresenta um exemplo de grafo G. As cliques máximas do grafo são formadas pelos conjuntos dos vértices $\emptyset, \{1, 2\}, \{1, 7\}, \ldots, \{2, 6, 7\}$. Uma clique máxima contém os vértices $\{1, 7, 2\}$ ou $\{2, 6, 7\}$. Uma clique é *maximal* se ela não é um subconjunto de uma clique maior. Neste grafo os conjuntos de vértices $\{1, 2, 7\}, \{2, 3\}, \{2, 6, 7\}$ e $\{2, 4, 5\}$ são as cliques maximais.

Figura 6 – Grafo G que contém um conjunto de cliques maximais



Definimos o problema computacional equivalente ao problema da CM como se segue.

Problema da Clique Máxima (CM)	
Instância:	Um grafo G.
Devolva:	Uma clique máxima de G.

Este é um problema já estudado por diversos autores. Em Bomze et al. (1999) diversos algoritmos de solução exata para o problema são citados. Ainda segundo este autor um dos primeiros algoritmos para enumeração de todas cliques máximas de um grafo está descrito em Harary e Ross (1957). Contudo, este não é um algoritmo *backtracking* como descrito neste trabalho. Os algoritmos que utilizam a técnica *backtracking* que foram propostos por Akkoyunlu (1973) e Bron e Kerbosch (1973) – algoritmo (BK) – na década de 1970 possivelmente sejam os primeiros.

Em uma solução utilizando *backtracking*, a primeira coisa a analisar é como construir o conjunto C_k de candidatos que compõem uma clique em um grafo G. Assim, $a = (a_1, \ldots, a_k)$ é o conjunto de vértices de uma solução possível se e somente se $a_1, \ldots, a_k$ é uma clique. Seguindo essa ideia é possível conceber um algoritmo que enumera todas as cliques de um grafo. Logo, o próximo passo é implementar um algoritmo que escolhe a maior clique dentre todas enumeradas.

O algoritmo Bron-Kerbosh(BK) é um algoritmo que *rapidamente* enumera todas cliques distintas de um grafo. Este algoritmo é descrito em duas versões. Na verdade, BK é um algoritmo que pode ser modificado facilmente dando origem a ramificações do mesmo. Sendo assim, uma base importante para novos algoritmos.

Para enumerar todas as cliques de um grafo sem repetições, BK utiliza três conjuntos disjuntos Q, K e S. O conjunto Q contém os vértices que fazem parte da clique atual. O conjunto K contém os vértices candidatos que podem ser adicionados à clique, ou seja, os vértices conectados a todos vértices em Q. Por fim, o conjunto S contém os vértices já processados, os quais não podem complementar Q, pois todas cliques contendo estes vértices já foram geradas.

O algoritmo BK1(Q, K, S) é a descrição da primeira versão de BK. Sua chamada é iniciada com os conjuntos $Q = \emptyset$ e $S = \emptyset$. O conjunto K deve iniciar com todos vértices do grafo. Quando K e S forem vazios o conjunto Q é uma clique maximal. O algoritmo afasta cliques repetidas utilizando o conjunto S através da atualização $K = K - \{u_i\}$ dado um vértice arbitrário $u \in V$. Um conjunto auxiliar N é definido como os vizinhos de um vértice u_i.

BK1(Q, K, S)

Entrada: vetor Q; vetor K; vetor S, vetor global N.
Saída: conjunto de todas cliques em um grafo.

```
1 Se K = ∅ e S = ∅ então
2 |   Devolva Q
3 Senão
4 |   Para i ← 1 até k faça
5 |   |   K ← K − {u_i}
6 |   |   N ← {v ∈ V : {u_i, v} ∈ E}
7 |   |   BK(Q ∪ {u_i}, K ∩ N, S ∩ N)
8 |   |   S ← S ∪ {u_i}
```

O algoritmo BK1(Q, K, S) decide se uma clique é maximal somente quando o conjunto Q é totalmente formado. A segunda versão do algoritmo BK tenta decidir se uma clique é maximal mais rapidamente pela aplicação de uma heurística na escolha de um vértice u_p chamado *pivô*. Isso pode implicar na diminuição de espaço de busca pela eliminação de estados que não levaram a uma clique maximal. Em outras palavras, existe uma poda no espaço de busca.

O algoritmo BK2(Q, K, S) é baseado no algoritmo BK1(Q, K, S). Uma heurística

possível para a escolha do pivô pode ser definida como a escolha dos vértices de maior grau em ordem crescente. A clique que contém um vértice de maior grau não significa ser a maior clique do grafo. Mas a escolha de um vértice $u_p \in P$ pode diminuir o tamanho do espaço de busca.

BK2(Q, K, S)

Entrada: vetor Q; vetor K; vetor S, vetor global N.
Saída: conjunto de todas cliques em um grafo.

```
1  Se K = ∅ e S = ∅ então
2      Devolva Q
3  Senão
4      Tomar um vértice u_p de P
5      Para i ← 1 até k faça
6          Se u_i não é adjacente a u_p então
7              K ← K − {u_i}
8              N ← {v ∈ V : {u_i, v} ∈ E}
9              BK(Q ∪ {u_i}, K ∩ N, S ∩ N)
10             S ← S ∪ {u_i}
```

O algoritmo BK ramifica vários outros algoritmos. Em Cazals e Karande (2008) é discutida a relação entre o trabalho proposto por Tomita, Tanaka e Takahashi (2006) e o trabalho apresentado por Koch (2001), os quais utilizam o algoritmo BK como base.

5.4 BRANCH AND BOUND

Para introduzir a técnica de *Branch and Bound* retomamos alguns conceitos já discutidos: um algoritmo backtracking examina todas soluções parciais de uma instância; uma poda em um árvore de estados tenta diminuir seu espaço de busca. Uma poda mais inteligente pode ser feita determinando limitantes superiores ou inferiores, seção 5.5. Um estado cujo a solução parcial ultrapassa esse limitante é podado, ou seja, a partir deste estado todos seus descendentes são desconsiderados. A técnica *Branch and Bound* consiste em examinar todas soluções parciais (passo *branching*) eliminando os estados que não podem levar a uma solução viável (passo *bounding*).

Em Züge (2011) são discutidos diversos algoritmos baseados em *Branch and Bound* para a solução da clique máxima de forma exata. Neste trabalho, é possível notar que partindo do algoritmo BK são concebidos diversos outros algoritmos, dos quais, podemos citar Tomita e Kameda (2007) que propõe o algoritmo MCQ e Tomita et al. (2010) que propõe o algoritmo MCS.

Pela modificação do algoritmo BK2(Q, K, S), o algoritmo **MaxCliqueBB** proposto em Carmo e Züge (2012) apresenta um algoritmo *Branch and Bound* como um *framework* de implementação. MaxCliqueBB apresenta algumas funções que podem ser especializadas para construção de vários algoritmos que resolvem o problema da clique máxima. Os algoritmos MCQ e MCS são exemplos de implementações utilizando este framework. Os resultados experimentais

com a implementação de diversos algoritmos sobre o framework MaxCliqueBB podem ser consultados neste mesmo trabalho. O algoritmo MCBB(G) descreve o algoritmo MaxCliqueBB.

MCBB(G)

Entrada: um grafo G.
Saída: maior clique de G.
1 $(C, P) \leftarrow$ pre-process(G)
2 **Enquanto** $P \neq \emptyset$
3 $\quad (Q, K) \leftarrow$ pre-process-state(G, pop(P), C)
4 $\quad$ **Enquanto** $K \neq \emptyset$ *e* $|C| < |Q|$+*bound(G,Q,K)*
5 $\quad\quad v \leftarrow$ remove(K)
6 $\quad\quad P \leftarrow$ push(G, Q, K)
7 $\quad\quad (Q, K) \leftarrow$ pre-process-state(G, $Q \cup \{v\}$, $K \cap N[v]$, C)
8 $\quad\quad$ **Se** $|C| < |Q|$ **então**
9 $\quad\quad\quad (C, P) \leftarrow$ update(G, C, P, Q)
10 **Devolva** *post-process*(G, C)

O conjunto C guarda a maior clique encontrada e o conjunto Q contém a clique atual. P é uma pilha (*last in first out*) de estados que implementa a recursão do algoritmo BK2. As funções de MCBB(G) podem ser vistas com mais detalhes em Carmo e Züge (2012). O passo *branching* corresponde a função remove(K) e o passo *bounding* corresponde a bound(G, Q, K). O vértice v retornado em remove(K) corresponde ao vértice pivô. A escolha adequada de um pivô através de uma heurística pode levar a um espaço de busca menor. A notação $N[v]$ corresponde aos vizinhos do vértice v.

5.5 PROBLEMA DA MOCHILA

O Problema da Mochila(PM) é um problema combinatório de otimização com diversas aplicações. O problema não aparenta ser de difícil entendimento, podendo ser introduzido utilizando diferentes abordagens e contextos. Nesta seção o PM será resolvido utilizando um algoritmo backtracking com limitante superior.

Em Kellerer, Pferschy e Pisinger (2003) são apresentados muitos campos onde este problema está presente. Existem exemplos de áreas como a economia e criptografia que demonstram suas aplicações práticas. Problemas relacionados a investimentos, transporte e empacotamento podem estar intimamente relacionados ao PM. Segundo o autor, sua versão de otimização é um problema $\mathcal{NP}$-difícil pelo menos tão difícil quanto sua versão de decisão. Assim, como o problema da clique máxima, sua versão de decisão é citada na lista de 21 problemas de decisão $\mathcal{NP}$-completos de Karp (1972).

Sendo PM um problema $\mathcal{NP}$-completo não é conhecido um algoritmo de tempo polinomial que resolva o problema. Em alguns sistemas de criptografia de chave pública, a utilização do PM é recorrente, pois se baseia no fato de que este é um problema $\mathcal{NP}$-completo e não existe algoritmo eficiente que o resolva. O sistema de criptografia Merkle-Hellman é um

exemplo de sistema que utiliza o PM. Apesar disso, utilizando outras características inerente ao PM, o sistema já foi quebrado, ver mais em Shamir (1984).

O Problema da Mochila é enunciado a seguir. Devido ao fato de existirem variações contextuais da apresentação do problema, durante este trabalho será enuncia mais de um contexto para o problema.

Problema da Mochila

Considerando uma mochila de capacidade W, dado um conjunto N que consiste em n itens i com lucro v_i e peso w_i, o objetivo é selecionar um subconjunto de N, tal que, a soma dos lucros associado a cada item seja a maior possível sem ultrapassar W.

Exemplo. Seja $W = 11$ e $n = 4$, os itens

item i	1	2	3	4
peso w_i	7	4	5	2
lucro p_i	31	15	17	10

O melhor lucro possível para esse exemplo pode ser calculado escolhendo um item de cada vez. Realizando as combinações entre os itens disponíveis o melhor lucro é 46 dos itens 1 e 2.

Durante este trabalho será frequente a referência do PM. Definimos o problema computacional equivalente como segue.

PROBLEMA DA MOCHILA(PM)

Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W.
Devolva:	uma n-upla $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, tal que,
	maximize $P = \sum_{i=1}^{n} v_i x_i$,
	sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$.

Uma solução possível para este problema pode ser baseada no algoritmo Subconjuntos(k) definido na seção 5.1. Definimos as seguintes funções e notações para auxiliar na implementação do algoritmo: um subconjunto $X = \{x_1, x_1, \ldots, x_n\}$ contendo os itens da n-upla sendo construída onde

$$P(X) \coloneqq \sum_{i=1}^{n} x_i v_i,$$

e

$$S(X) \coloneqq \sum_{i=1}^{n} x_i w_i$$

são funções definidas para auxiliar na construção da resposta algorítmica. Definimos o algoritmo backtracking M(k) sem nenhum tipo de poda. A chamada inicial do algoritmo define $k = 1$. Por ser baseado no algoritmo Subconjuntos(k) este algoritmo tem complexidade $O(n2^n)$.

M(k)

Entrada: inteiro k; vetor global C_k; vetor global X.
Saída: lucro máximo.

```
1  Se k > n então
2  |  Se S(X) ≤ W então
3  |  |  Se P(X) > P então
4  |  |  |  P = P(X)
5  |  |  |  X = {x_1, ..., x_n}
6  Senão
7  |  C_k = {0, 1}
8  |  Para c ∈ C_k faça
9  |  |  x_k = c
10 |  |  M(k + 1)
```

Uma forma de podar o espaço de busca para o algoritmo M(k) é pela definição de um limite superior definido por uma função B. Essa ideia pode ser generalizada para problemas que buscam por um valor ótimo (máximo ou mínimo). Utilizando as notações definidas anteriormente, P(X) retorna o valor ótimo de uma solução parcial a partir $X = \{x_1, \ldots, x_k\}$ até seus descendentes X'.

Uma função *bounding* pode ser interpretada como aproximação de P(X). Um algoritmo de uso geral que utiliza essa ideia pode ser descrito. O algoritmo Bounding(k) é uma modificação do algoritmo Backtracking(K).

Para evitar que P(X) seja calculado sucessivas vezes para valores que não levam a uma solução ótima é utilizada uma função B. Uma função *bounding* é um valor real definida sobre um estado da árvore de estados, satisfazendo a condição B(X) $\geq$ P(X) para problemas de maximização ou B(X) $\leq$ P(X) para problemas de minimização onde X é qualquer solução parcial. (KREHER; STINSON, 1999)

A ideia de bounding está presente em algoritmos, como sugere o nome, *Branch and Bound*. Uma função bounding pode ser difícil de se obter.

Para o PM, considere a seguinte variação do problema: completar a mochila no caso que os itens podem ser fracionados. Neste caso, é possível "quebrar" um item e seu valor será proporcional o valor do item inteiro. Este novo problema pode ser resolvido ordenando-se os itens pela razão v/w, onde

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n},$$

e escolhendo os ítens de maior lucro, até que a mochila esteja cheia. Os primeiros ítens são inseridos por completo na mochila, e no máximo um será fracionado. A resposta para esse

```
Bounding(k)
    Entrada: inteiro k; vetor global C_k; vetor global X.
    Saída: lucro máximo.
1   Se X é uma solução viável então
2       Se P(X) > P então
3           P = P(X)
4           X = {x_1, ..., x_k}
5   Senão
6       C_k = candidatos
7       B = B(X)
8       Para c ∈ C_k faça
9           Se B ≥ P(X) então
10              x_k = c
11              Bounding(k + 1)
```

novo problema não é necessariamente igual ao da mesma instância para PM, mas pode ser maior. Ou seja, o processo descrito é um limitante superior para o PM.

Além da técnica de backtracking utilizando uma função $B(\cdot)$ como limitante, este problema pode ser resolvido utilizando a técnica de *Branch and Bound*. Além disso, existem casos particulares do problema que podem ser resolvidos utilizando Algoritmos Gulosos (MAGAZINE; NEMHAUSER; JR, 1975). Porém, este é um problema de otimização que carrega uma estrutura recursiva aproveitada na técnica de programação dinâmica. Sendo assim, uma abordagem de resolução por programação dinâmica é discutida na seção 6.2.1 que retoma o problema visando a construção de uma solução mais eficiente. Em Kellerer, Pferschy e Pisinger (2003) e Martello e Toth (1990) são apresentadas algumas versões de PM como casos especiais. Um dos casos especiais tratados é o Problema das Moedas. Na seção 6.2.3 o problema das moedas é resolvido pela técnica de programação dinâmica por ser uma técnica que garante algoritmos mais eficientes em comparação a algoritmos de backtracking.

O capítulo 6 trata da apresentação da técnica de Programação Dinâmica. Neste capítulo são abordados problemas de otimização para os quais uma solução por força bruta não é eficiente.

6 PROGRAMAÇÃO DINÂMICA

Em campos de estudo como os da Matemática, Pesquisa Operacional e Ciência da Computação, existem problemas nos quais é preciso encontrar a melhor solução possível dentre um conjunto de respostas viáveis. Este tipo de problema, sugere explorar todas soluções pela decomposição do problema original em problemas menores ou *subproblemas*. A programação dinâmica é um caminho para encontrar algoritmos eficientes para problemas que podem ser quebrados em problemas menores. De forma geral, a técnica de programação dinâmica fornece um modelo de abordagem na resolução de problemas.

Essa técnica pode ser comparada a outras técnicas de solução de problemas. Algoritmos que empregam a estratégia *dividir e conquistar*, resolvem problemas ao particionar o problema original em problemas menores ou subproblemas, resolvendo recursivamente os subproblemas, e em seguida, combinando as soluções para resolver o problema original. Em programação dinâmica aplicamos a mesma estratégia – resolver todos os subproblemas – e armazenamos as respostas para usá-las na resolução do problema original. (SEDGEWICK, 1990). A técnica de fato se assemelha com o paradigma de divisão e conquista, mas tende a ser eficiente uma vez que armazena as soluções menores e evita computar o mesmo subproblema repetidas vezes.

Programação dinâmica é uma técnica para implementar um programa recursivo eficiente guardando resultados parciais. Uma vez que algoritmos recursivos resolvem o mesmo subproblema repetidas vezes, podemos armazenar a solução de cada um desses subproblemas em uma tabela e consultá-la ao invés de resolver novamente o mesmo subproblema.(SKIENA, 1998)

O exemplo do algoritmo que calcula o n-ésimo termo da sequência de Fibonacci é utilizado como introdução dos conceitos iniciais que envolvem uma solução por programação dinâmica, discutido na Seção 6.1. Considerando os conceitos presentes na Seção 6.1 gostaríamos de resolver problemas de forma eficiente. A Seção 6.2 apresenta uma sequência de etapas sugeridas para a resolução de problemas de otimização pela técnica de programação dinâmica. As Seções 6.2.1, 6.2.2, 6.2.3, 6.2.4 e 6.2.5, apresentam problemas exemplares que podem ser resolvidos de maneira eficiente com a aplicação dos conceitos de programação dinâmica.

6.1 SEQUÊNCIA DE FIBONACCI

A ideia básica de programação dinâmica é melhor entendida através de exemplos. Os Números de Fibonacci ou Sequência de Fibonacci é um exemplo prático da aplicação da técnica. Esta trata da aplicação da técnica de programação dinâmica que otimiza o algoritmo que calcula o n-ésimo termo da sequência de Fibonacci.

6.1.1 Sequência de Fibonacci por recursão

A sequência de números proposta pela matemático italiano Fibonacci no século XIII foi originalmente proposta para modelar o crescimento populacional hipotético de coelhos. Fibonacci supôs que o número de pares de coelhos nascidos em dado mês, era igual o número de pares de coelhos nascidos em cada um dos últimos dois meses, partindo de um par de coelhos no primeiro mês, sendo que, os coelhos nunca morrem. Para contar o número de coelhos nascidos no n-ésimo mês, ele definiu uma relação de recorrência que usualmente é definida como

$$F_n = \begin{cases} 0 & \text{se n} = 0 \\ 1 & \text{se n} = 1 \\ F_{n-1} + F_{n-2} & \text{se n} > 1 \end{cases}$$

Assim, os primeiros termos da sequência são apresentados $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots)$. Partindo da relação de recorrência, podemos escrever um algoritmo recursivo para calcular o n-ésimo termo da sequência no algoritmo Fib(n).

Fib(n)

Entrada: inteiro n.
Saída: n-ésimo termo da sequência.

```
1 Se n = 0 então
2 |  Devolva 0
3 Se n = 1 então
4 |  Devolva 1
5 Devolva Fib(n − 1) + Fib(n − 2)
```

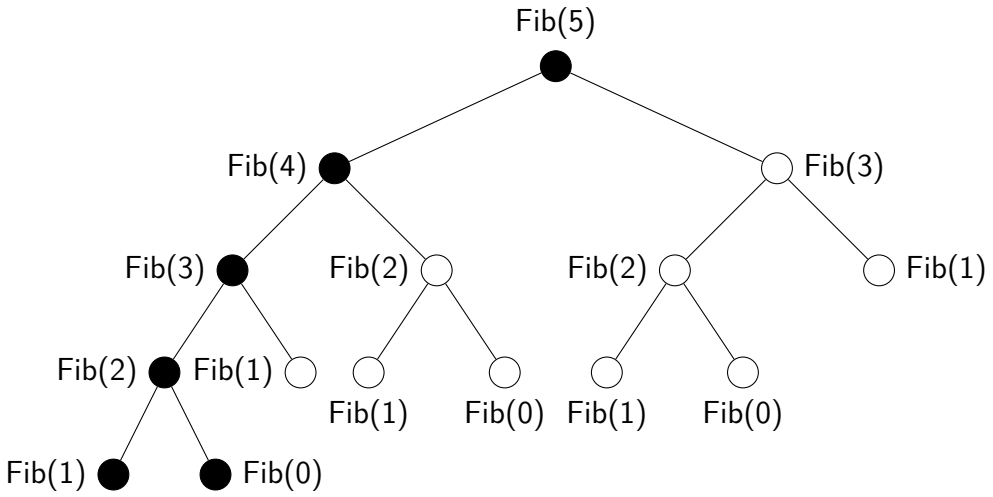
Ao executar a função Fib, podemos interpretar suas chamadas recursivas como uma árvore, ilustrada na Figura 7. A árvore foi gerada a partir da busca pelo 5º elemento na sequência.

Note que Fib(2) é calculado várias vezes em ambos os lados da árvore de recursão. Note também, que Fib(1) de forma análoga é calculado sucessivas vezes, nesse caso o mesmo subproblema é calculado 5 vezes. Essa característica do procedimento recursivo para a sequência de Fibonacci é ineficiente, na verdade, o algoritmo tem $\Theta(1.6^n)$ número de chamadas, portanto complexidade $\Theta(1.6^n)$.

6.1.2 Sequência de Fibonacci por Programação Dinâmica

A programação dinâmica, é o método que deve ser empregado caso seja observado que uma solução recursiva *ingênua* é ineficiente, com a condição de que um mesmo subproblema é resolvido repetidas vezes. Assim, remodelamos para que cada subproblema seja resolvido apenas uma vez, **memoizando** essa solução. Dessa forma, se for preciso localizar a

Figura 7 – Árvore de recursão sobre o algoritmo Fib(n). Os nós marcados com branco são chamadas já feitas pelo algoritmo em outro momento.



solução de um subproblema novamente em outro momento, apenas o consultamos ao invés de resolver novamente. Portanto, a programação dinâmica usa memória adicional para salvar soluções, caracterizando um exemplo de troca de **espaço** por **tempo** — uma solução de tempo exponencial pode ser transformada em uma solução de tempo polinomial. (CORMEN, 2009).

6.1.3 Tipos de Abordagem

A programação dinâmica pode ser aplicada por duas abordagens devido a natureza dos problemas os quais a técnica se aplica. As abordagens para implementar um procedimento em programação dinâmica são as abordagens ***top-down*** com **memoização** e o método ***bottom-up***.

Uma abordagem *top-down* é uma forma de implementar um procedimento de maneira natural, ou seja, resolve os casos mais distante do caso base, chamando a si mesmo enquanto não chaga na base. O procedimento sempre verifica se já resolveu previamente um subproblema. Se sim, retorna a resposta salva (frequentemente salva em um vetor ou tabela *hash*), evitando o recalculo dos subproblemas a partir desse instante. Caso o subproblema não tenha sido resolvido previamente, o procedimento o resolve e salva a solução. Dizemos que o procedimento recursivo é *memoizado* se ele é capaz de "lembrar" resultados anteriores. (CORMEN, 2009)

A outra abordagem, *bottom-up*, é um método contrário a abordagem *top-down*. Resolvemos primeiro os subproblemas menores e a partir deles resolvemos os subproblemas maiores, sendo que os subproblemas dependem da resolução de um menor. Para tanto, partindo do caso base, resolvemos todos os subproblemas mais a cima salvando cada solução. Resolvemos exatamente uma vez cada subproblema. (CORMEN, 2009)

6.1.4 Fibonacci pela abordagem *top-down*

Na sequência de Fibonacci podemos aplicar a ideia de **memoizar** as soluções para $F(k)$ em uma tabela indexada pelo parâmetro k. Para evitar resolver um mesmo subproblema k é preciso verificar se ele foi ou não resolvido anteriormente. (SKIENA, 1998)

O procedimento a seguir resolve o problema de encontrar o n-ésimo termo da sequência por uma abordagem *top-down* com memoização, salvando a solução de cada subproblema caso ele não tenha sido resolvido antes, partindo dos subproblemas mais distante do caso base.

Fib(n)

Entrada: inteiro n.
Saída: n-ésimo termo da sequência.
1 Tomar $memo[0 \ldots n]$ como novo vetor
2 memo $[0] \leftarrow 0$
3 memo $[1] \leftarrow 1$
4 **Para** $i = 2$ **até** n **faça**
5 | memo $[i] \leftarrow -1$
6 **Devolva** `Fib-Memo`(n, *memo*)

Fibi-memo(n, memo)

Entrada: inteiro n; vetor memo.
Saída: vetor memo.
1 **Se** memo $[n] = -1$ **então**
2 | memo $[n] \leftarrow$ `Fib-Memo`$(n-1, memo)$ + `Fib-Memo`$(n-2, memo)$
3 **Devolva** memo $[n]$

Para memoizar as soluções foi usado um vetor chamado *memo* que armazena o resultado para F_n. Em Fib(n) inicializamos memo com os casos base nas linhas 2 e 3, marcamos que memo ainda não guarda o valor de F_n para memo de 2 até n nas linhas 4 e 5 e, por fim, devolvemos o valor da função Fib-Memo(n, $memo$), que calcula F_n se ainda não foi calculado e armazenado em memo. A Figura 8 representa a árvore de recursão produzida pelo algoritmo Fib(n) aplicando a técnica de memoização.

6.1.5 Fibonacci pela abordagem *bottom-up*

Em uma abordagem *bottom-up* consideramos primeiro os casos base da relação de recorrência. A partir do caso base expandimos, resolvendo os problemas maiores a partir da solução dos menores já resolvidos. O procedimento Fib(n) a seguir considera primeiro o caso base da relação de recorrência – pela definição $F(0) = 0$ e $F(1) = 1$ – nas linhas 2 a 3 e resolve os próximos termos da sequência, salvando todas soluções, até resolver F_n.

Assim, eliminamos todas chamadas recursivas. Esse algoritmo tem complexidade $O(n)$, o que é mais eficiente que o algoritmo recursivo Fib(n) da seção 1.1.

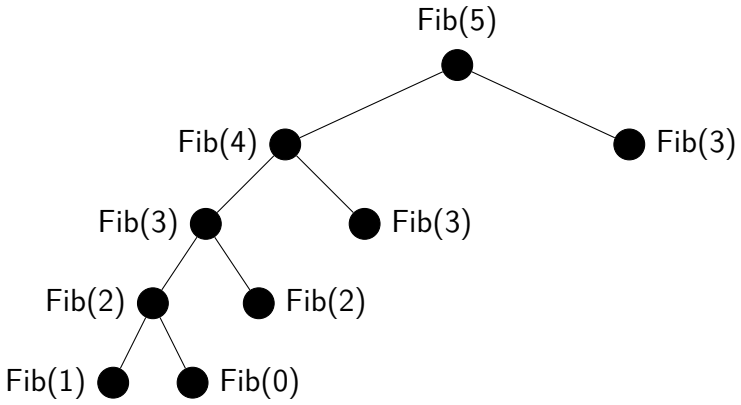


Figura 8 – Árvore de recursão sobre o algoritmo Fib(n) com memoização.

Fib(n)

Entrada: inteiro n.
Saída: n-ésimo termo da squência.
1 Tomar $f[0...n]$ como um novo vetor
2 $f[0] = 0$
3 $f[1] = 1$
4 **Para** $i = 2$ **até** n **faça**
5 $\quad f[i] = f[i-1] + f[i-2]$
6 **Devolva** $f[n]$

6.2 RESOLVENDO PROBLEMAS COM PROGRAMAÇÃO DINÂMICA

A programação dinâmica, em geral, é aplicada em *problemas de otimização*, definidos na seção 3.5.1. Problemas que procuram uma solução ótima podem ser formados por muitas soluções possíveis, cada solução apresenta um valor e o objetivo é a solução de valor ótimo (máximo ou mínimo). Essa solução é chamada de **uma** solução ótima, ao invés de **a** solução ótima, pois pode existir várias soluções que alcançam o valor ótimo. (CORMEN, 2009)

O termo Programação Dinâmica foi cunhado por Richard Bellman ainda na década de 1950 enquanto trabalhava para *RAND Corporation* (BELLMAN, 1952; BELLMAN, 1954). Em Dreyfus (2002) existe um apanhado histórico do caminho trilhado por Bellman para cunhar o termo e iniciar as primeiras obras. A palavra "Programação" refere-se ao planejamento de uma solução ótima, em forma de tabela por exemplo. A palavra "Dinâmica" diz a característica dinâmica ou vários estágios, variação. Por tanto, programação dinâmica é a criação de um planejamento de forma otimizada de um processo de vários estágios (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006). A obra de Bellman (2013) introduz o campo de uma maneira extensiva.

Uma ideia básica é iniciar a resposta do problema e procurar por uma solução recursiva ingênua, ou seja, uma função recursiva de tempo exponencial, a qual divide o problema original em subproblemas menores e combina as soluções até resolver o problema. A partir da função

recursiva, melhoramos o algoritmo salvando as soluções dos subproblemas e aplicamos uma segunda melhoria resolvendo o problema em uma *ordem* mais eficiente. Essas ideias apareceram pela primeira vez nas seções 6.1.4 e 6.1.5, quando aplicamos a técnica sobre algoritmo que calcula o n-ésimo termo da sequência de Fibonacci. Nesta seção desejamos generalizar ainda mais esse conceito, resolvendo problemas de otimização. Porém, a técnica de programação dinâmica pode ser aplicada a muitos tipos de problemas distintos.

O desenvolvimento de um algoritmo de programação dinâmica pode ser divido em uma sequência de quatro etapas:

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima.
4. Construir uma solução ótima a partir de informações calculadas.

As etapas 1, 2 e 3 formam a base de uma solução de programação dinâmica para um problema. A etapa 4 é necessária se o valor de uma solução ótima é solicitado.(CORMEN, 2009)

6.2.1 Problema da Mochila

O Problema da Mochila já foi resolvido na seção 5.5. Sabendo que esse é um problema $\mathcal{NP}$-difícil, não é esperado uma solução eficiente para o problema. Contudo, é possível implementar uma solução com complexidade de tempo melhor em comparação com a soluções apresentadas até aqui. De fato, a aplicação da técnica de programação dinâmica fornece uma solução mais eficiente do que respostas anteriores. Com programação dinâmica, a solução para o Problema da Mochila tem complexidade pseudo-polinomial $\Theta(nW)$.

Retomando o Problema da Mochila(PM), este é um problema combinatório clássico onde se espera uma solução ótima. Dado um conjunto N composto por $1, \ldots, n$ itens com lucro v_i e peso w_i. O objetivo é selecionar um subconjunto tal que a soma dos lucros de seus elementos seja a maior possível. O PM é um problema de muitas aplicações, várias delas descritas em Kellerer, Pferschy e Pisinger (2003) envolvendo carregamento de cargas, empacotamento e investimentos de capital.

Exemplo. Seja o $W = 11$, $n = 4$ itens

item i	1	2	3	4
lucro v_i	31	15	17	10
peso w_i	7	4	5	2

uma solução ótima é dada pelo lucro de 46, utilizando uma combinação com os itens 1 e 2. Para esta instância, essa é a única solução ótima possível de ser encontrada.

Nas Seções 3.3.1 e 5.5 foi utilizada uma definição formar do problema computacional do PM. A definição formal é retomada como se segue.

Problema da Mochila(PM)	
Instância:	*lucros* $v_1, v_2, \ldots, v_n$;
	pesos $w_1, w_2, \ldots, w_n$;
	capacidade W.
Devolva:	uma n-upla $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, tal que,
	maximize $P = \sum_{i=1}^{n} v_i x_i$,
	sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$.

O algoritmo M(k) proposto na Seção 5.5 dá uma solução em tempo exponencial, $O(2^n)$. O algoritmo enumera todos 2^n subconjuntos de itens e verifica qual é a solução ótima. Um algoritmo com esse tempo de execução é impraticável em instâncias onde o n é grande. Observando esse algoritmo é possível notar que uma solução ótima é obtida por uma sequência de escolhas sobre uma instância.

Na Seção 3.2 o coeficiente binomial, denotado $\binom{n}{k}$ é interpretado como a escolha de k objetos dentre n. No PM, não é possível saber de imediato qual o valor de k para que a saída seja uma solução ótima, por isso, calcula-se todos valores para k onde $0 \leq k \leq n$. Essa é uma propriedade do triângulo de Pascal, definida $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. Para algum n e k não negativo, $\binom{n}{k}$ é calculado pela recorrência 6.1, dada por

$$\begin{aligned} \binom{n}{0} &= \binom{n}{n} = 1 \\ \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} \end{aligned} \tag{6.1}$$

Essas relações ajudam a resolver o PM por programação dinâmica. As etapas da Seção 6.2 são como um roteiro para a construção de um algoritmo eficiente.

Caracterizando uma solução ótima

Um dos princípios centrais para a resolução de um problema pela técnica de programação dinâmica passa por expressar uma solução em termo de subproblemas. Sendo assim, surge a pergunta, como é possível expressar o problema da mochila em termos de problemas menores? Considerar toda capacidade $w \leq W$ é uma forma de expressar um problema menor que o original, além disso, considerar os itens $1, \ldots, i$ para $i \leq n$. A função $f(W, i)$ é definida como

$$f(W, i) = \text{valor máximo possível usando a capacidade W e itens } 1, \ldots i.$$

Considerando uma solução ótima para o PM, a remoção de algum item r dessa solução implica em que o conjunto restante deve ser uma solução ótima para o subproblema definindo pela capacidade $W - w_r$ e o conjunto $N - \{r\}$. Essa propriedade diz que uma solução ótima é definida em termos de soluções ótimas de subproblemas. Nesses casos, é dito que um problema exibe uma condição de **subestrutura ótima**. Essa característica é um requisito fundamental para a aplicação da técnica de programação dinâmica.

Solução recursiva

Para expressar f recursivamente, de maneira indutiva, existem dois casos:

Carregar um item: Levar o item i resulta no lucro v_i, mas capacidade w_i ocupada. Com isso, resta $W - w_i$ de capacidade na mochila e o subconjunto de $1, 2, \ldots, i-1$ itens. Isso pode ser expresso como $v_i + f(W - w_i, i-1)$.

Deixar um item: Ao deixar um item para trás durante o saque, a capacidade W não se altera restando o subconjunto de itens $1, 2, \ldots, i-1$, isto é, $f(W, i-1)$.

Existem somente estas duas possibilidades de ação para escolher o subconjunto de itens que satisfaça as condições do problema, concebendo a seguinte equação de recorrência

$$f(W,i) = \begin{cases} 0 & \text{se } W, i = 0 \\ f(W, i-1) & \text{se } w_i > W \\ \max(f(W,i-1), p_i + f(W - w_i, i-1)) & \text{se } w_i \leq W \end{cases} \tag{6.2}$$

A recorrência 6.2 é chamada *recursão da programação dinâmica*, também conhecida como *equação de Bellman*. Em geral uma solução de programação dinâmica é construída a partir da definição de uma recorrência.

Calcular uma solução ótima

A equação 6.2 é uma solução recursiva para o problema. Descrevemos o algoritmo recursivo MR(W, i) sobre $f(W, i)$ seguindo uma abordagem *top-down*.

O algoritmo MR(W,i), apenas implementa a recorrência da função f. A chamada inicial do algoritmo é dada pelo valor da capacidade W e $i = n$ itens. Essa abordagem, ainda não implementa um algoritmo eficiente, uma vez que esse é um algoritmo recursivo ingênuo, de complexidade $O(2^n)$. A Figura 9 é árvore de estados do algoritmo MR(W, i) sobre três itens de lucro $v_1 = 100, v_2 = 70, v_3 = 50$, pesos $w_1 = w_2 = w_3 = 2$ e capacidade $W = 4$. Aqui, os nós da árvore são a interpretados como subproblemas. O algoritmo recursivo é ineficiente pois calcula o mesmo subproblema mais de uma vez.

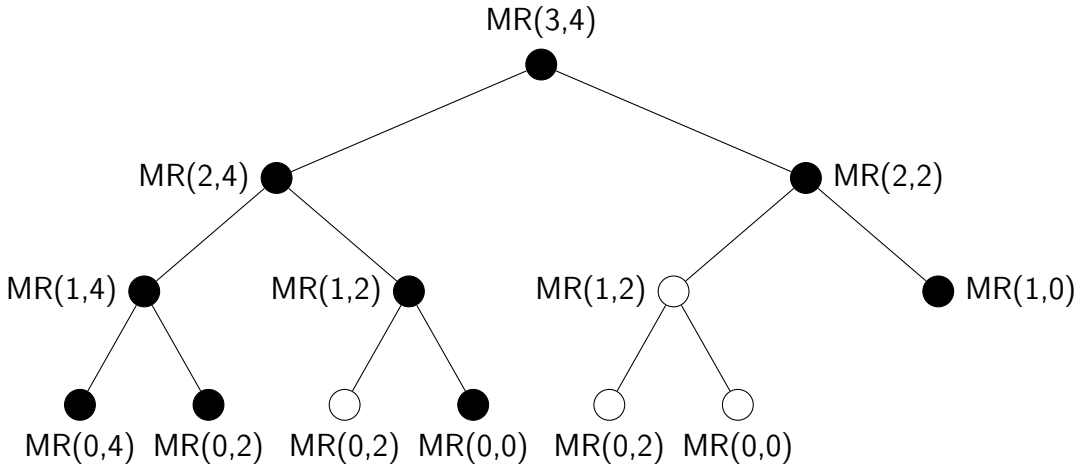
Quando um subproblema é resolvido repetidas vezes, dizemos que o problema de otimização apresenta a característica de **subproblemas sobrepostos**. Dessa forma, é possível

MR(W, i)

Entrada: inteiro W; inteiro i; vetor global v; vetor global w.
Saída: lucro máximo.

```
1 Se W = 0 ou i = 0 então
2 |  Devolva 0
3 Se w[i] > W então
4 |  Devolva MR(W, i − 1)
5 Senão
6 |  Devolva max(MR(W, i − 1), v[i]+ MR(W − w[i], i − 1))
```

Figura 9 – Árvore de estados sobre o algoritmo MR(W, i). Cada estado representa uma chamada do algoritmo para um problema menor. Os nós marcados como branco são subproblemas já resolvidos em algum momento anterior.



melhorar esse algoritmo aplicando a técnica de memoização, guardando a resposta dos subproblemas em um vetor ou matriz e recuperando essa resposta quando necessário, evitando calcular o mesmo subproblema.

Seja uma matriz $M[0..n, 0..W]$. Considerando i e j, onde $1 \leq i \leq n$ e $0 \leq j \leq W$, a entrada $M[i, j]$ guarda o valor máximo de um subconjunto $1, \ldots, i$ de itens que não ultrapasse a capacidade j. Não é possível saber imediatamente qual a entrada retorna o valor máximo, por isso calcula-se todas possíveis, fazendo $M[n, W]$ conter o valor máximo de n itens em uma mochila de capacidade W.

Aplicamos a técnica de programação dinâmica para obter uma solução recursiva mais eficiente. A função recorrente 6.2 fornece um algoritmo de programação dinâmica em uma abordagem *top-down*. O algoritmo MM(W,i) implementa a estratégia de memoizar

subproblemas utilizando uma matriz $M[0..n, 0..W]$.

MM(W, i)

Entrada: inteiro W; inteiro i; vetor global v; vetor global w.
Saída: lucro máximo.

```
1 Se W = 0 ou i = 0 então
2 |  Devolva 0
3 Se M[W, i] > −1 então
4 |  Devolva M[W, i]
5 Se w[i] > W então
6 |  M[W, i] ← MM(W, i − 1)
7 Senão
8 |  M[W, i] ← max(MM(W, i − 1), v[i]+ MM(W − w[i], i − 1))
9 Devolva M[W, i]
```

Modificando a notação que foi usada ao definir a função $f(W, n)$ que retorna uma solução ótima, dado dois inteiros i $(1 \leq i \leq n)$ e j $(0 \leq j \leq W)$, consideramos os subproblemas com itens $1, \ldots, i$ e capacidade j, denotamos $M[i, j]$ como solução ótima, definida como

$$M[i,j] = \begin{cases} 0 & \text{se } i, j = 0 \\ M[i-1, j] & \text{se } w_i > j \\ \max(M[i-1, j], p_i + M[i-1, j-w[i]]) & \text{se } w_i \leq j \end{cases} \tag{6.3}$$

Escrevemos o algoritmo MDP(n,W) em seguida. O algoritmo é uma implementação da função 6.9 em uma abordagem *bottom-up*. O Anexo A.2 disponibiliza uma implementação do algoritmo MDP(n, W) em linguagem de programação C++.

MDP(n, W)

Entrada: inteiro W; inteiro i; vetor global v; vetor global w.
Saída: lucro máximo.

```
1  Tomar M[0..n, 0..W] como uma nova matriz
2  Para i = 0 até n faça
3  |  M[i, 0] = 0
4  Para j = 0 até W faça
5  |  M[0, j] = 0
6  Para i = 1 até n faça
7  |  Para j = 0 até W faça
8  |  |  Se w[i] > j então
9  |  |  |  M[i, j] = M[i − 1, j]
10 |  |  Senão
11 |  |  |  M[i, j] = max(M[i − 1, j], v[i]+M[i − 1, j − w[i]])
12 Devolva M[n, W]
```

No algoritmo MDP(n,W) são realizadas operações sobre uma matriz $M[n, W]$ o que implica em no máximo $n \times W$ operações, ou seja, o algoritmo é $O(nW)$. A análise de um algoritmo deve ser feita sobre o tamanho sua entrada, como visto na Capítulo 4. A entrada n representa o tamanho de um conjunto com $1, 2, \ldots, n$ itens. O parâmetro W requer $b = \lfloor \log_2 W \rfloor + 1$ bits para ser representado, ou seja, o tamanho de W é dado por 2^b. Portanto, o algoritmo tem complexidade de tempo exponencial $\Theta(n2^{\log_2 W})$. Dizemos que o algoritmo em programação dinâmica para o PM tem complexidade **pseudo-polinomial** $\Theta(nW)$.

Problema Soma de Subconjunto

O Problema da Soma de Subconjunto é um problema que pode ser modelado sobre o Problema da Mochila, caracterizando um *caso particular* do problema da mochila. O problema da Soma de Subconjunto é enunciado em seguida, adotando a notação em Kozen (2012).

Soma de Subconjunto

Dado um conjunto A de n inteiros a_i e um inteiro W, existe um subconjunto, tal que, $\sum_{i \in A} a_i = W$?

Exemplo. Suponha o conjunto $A = \{3, 4, 8, 5\}$ existe um subconjunto de M cuja soma dos seus elementos é 17? Para esse exemplo, ao somarmos os elementos $5, 4, 8$, obtemos a soma 17.

Uma instância do Problema da Mochila, assumindo $v_i = w_i$ para todo $1 \leq i \leq n$, da instância a uma instância para problema Soma de Subconjuntos. Portanto, o problema da soma de subconjunto é o problema que maximiza

$$\sum_{i=1}^{n} w_i x_i$$

sujeito a condição

$$\sum_{i=1}^{n} w_i x_i = W,$$

tal que, x_i é *verdadeiro* se o i-ésimo item faz parte de uma solução ótima ou *falso*, caso contrário. Formalmente, o problema é definido como se segue.

Problema da Soma de Subconjunto	
Instância:	*conjunto* A;
	$w_i = a_i$;
	inteiro W
Devolva:	"sim" se existe subconjunto $A' \subset A$, tal que,
	$P = \sum_{i=1}^{n} w_i x_i$
	sujeito a $\sum_{i=1}^{n} w_i x_i \leq W$
	e $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$; "não", caso contrário.

Por ser esse, um caso especial do PM, a solução por programação dinâmica da origem a uma solução recursiva eficiente (MARTELLO; TOTH, 1990). Para tanto, o primeiro passo é a definição de uma solução em termos de uma função recursiva. Definimos uma função $S[i, j]$ como a função que devolve "verdadeiro" se existe subconjunto, tal que, $\sum_{i \in A} w_i = W$, e "falso" caso contrário. Assim como no PM, existem duas ações possíveis para cada elemento de A que levam à solução do problema.

Incluir elemento: Selecionar um elemento que faz parte da solução resulta em $W - w_i$ restando um subconjunto de elementos em $1, 2, \ldots, i-1$ de S onde $1 \leq i \leq n$, ou seja, $S[i-1, j-w_i]$.

Não incluir elemento: Quando não incluímos um item na solução, estamos apenas ignorando o item sem alterar W, ou seja, $S[i-1, j]$.

Para que $S[i, j]$ seja uma função recursiva é preciso definir os casos base da recorrência. Assim, a função é definida como

$$S[i,j] = \begin{cases} \text{Verdadeiro} & \text{se } j = 0 \\ \text{Falso} & \text{se } i = 0 \\ S[i-1, j] & \text{se } w_i > j \\ S[i-1, j] \vee S[i-1, j-w_i] & \text{caso contrário} \end{cases} \tag{6.4}$$

A recorrência 6.4 retorna "verdadeiro" caso exista um subconjunto que somado obtém W ou "falso", caso contrário.

Após estabelecer uma solução recursiva, o próximo passo é **(i)** implementar um algoritmo definido por 6.4; **(ii)** adicionar memoização para melhorar o tempo de complexidade; **(iii)** converter a solução em um algoritmo iterativo *bottom-up*. Da recorrência $S[i, j]$, é possível escrever um algoritmo de programação dinâmica. O algoritmo SomSubconjuntos(n,W) está descrito como segue. Retomando o algoritmo MDP(n,W) que resolve o PM, é possível notar que ele foi alterado o para resolver o problema da Soma de Subconjunto.

SomaSubconjunto(n, W)

Entrada: inteiro n; inteiro W; vetor global w.

Saída: soma de um subconjunto.

```
1  Tomar S[0..n, 0..W] como uma nova matriz
2  Para i ← 0 até n faça
3  |  S[i, 0] ← 1
4  Para j ← 0 até W faça
5  |  S[0, j] ← 0
6  Para i ← 1 até n faça
7  |  Para j ← 0 até W faça
8  |  |  Se w[i] > j então
9  |  |  |  S[i, j] ← S[i − 1, j]
10 |  |  Senão
11 |  |  |  S[i, j] ← max(S[i − 1, j],S[i − 1, j − w[i]])
12 Devolva S[n, W]
```

6.2.2 Problema da Partição

O Problema da Partição(PP) é um dos problemas clássicos definido como $\mathcal{NP}$-completo. Dado um conjunto $A = \{a_1, a_2, \ldots, a_n\}$ de inteiros, uma partição de A em dois subconjuntos A_1 e A_2 respeita as condições $A_1 \cup A_2 = A$, $A_1 \cap A_2 = \emptyset$. A função $w(A) = \sum_{j \in A} j$ devolve o *peso* de A. A versão de decisão do PP é o problema que questiona: é possível particionar A em 2 subconjuntos onde o peso de cada subconjunto seja igual? Esse problema pode ser modelado como um problema de otimização: dado o conjunto A, encontrar dois conjuntos A_1, A_2, tal que, a diferença entre o peso máximo de cada subconjuntos seja minimizado. (KELLERER et al., 1997)

O PP é um problema citado na lista de 21 problemas de decisão $\mathcal{NP}$-completos de Karp (1972). É interessante pois a partir dele é possível provar o caráter $\mathcal{NP}$-completo do PM e seus casos particulares. As reduções do PP em Problema da Mochila e Problema das Moedas provam que estes de fato são problemas $\mathcal{NP}$-completo, ver mais em Martello e Toth (1990).

O Problema da Partição é relacionado ao problema do escalonamento de tarefas em computadores iguais. Nesse caso, o contexto é dado sobre k máquinas paralelas idênticas que recebem um conjunto disjunto de n tarefas. O objetivo é minimizar o tempo para processar todas as tarefas. (BARUAH; FISHER, 2005)

Problema da Partição

Dado o conjunto A, decidir se é possível particioná-lo em subconjuntos A_1, A_2 de modo que a soma dos elementos dos subconjuntos sejam iguais.

Exemplo. Seja o conjunto $A = \{1, 1, 2\}$, definir se existem subconjuntos A_1, A_2 tal que $A_1 \cap A_2 = A$, $A_1 \cup A_2 = \emptyset$. Nesse exemplo a resposta é "sim" onde $A_1 = \{1, 1\}$ e $A_2 = \{2\}$.

Dizemos que este é o problema 2-partição, um caso especial do problema k-partição, para um $k > 0$. O problema da k-partição é o problema que generaliza o PP. Definimos o problema computacional em seguida.

Problema da Partição	
Instância:	conjunto A;
	inteiro k;
Devolva:	"sim" se existe $A' \subseteq A$, tal que,
	minimize $M(A_1, A_2, \ldots, A_k) = max\{\sum_{j \in A_i} j\} - min\{\sum_{j \in A_i} j\}$;
	"não", caso contrário.

Existem na literatura algumas estratégias e algoritmos para a resolução do PP. Em Horowitz e Sahni (1974) são discutidas várias estratégias de resolução, dentre elas, *Branch and Bound* e programação dinâmica. Outro método de resolução passa pela aplicação de uma estratégia gulosa para a resolução do problema (KORF, 1998). O algoritmo Karmark-Karp, resolve o problema aplicando uma heurística mais aprimorada (chamada *diferença de conjunto*) do que a empregada nas soluções gulosas, ver mais em Karmarkar e Karp (1982).

O PP permite uma resolução simples aproveitando a relação com o problema da soma de subconjuntos da Seção 6.2.1. Uma implementação com alterações triviais no algoritmo SomaSubconjunto(W,n) concebe o algoritmo Partition(n, W) definido como segue. Esse algoritmo tem complexidade de tempo pseudo-polinomial $O(nW)$. O Anexo A.3 disponibiliza uma implementação do algoritmo Partition(n, W) em linguagem de programação C++.

Partition(n, W)

Entrada: inteiro n; inteiro W; vetor global w.

Saída: soma obtida ao particionar um conjunto.

```
1  s ← 0
2  Para i ← 0 até n faça
3  |  s ← s + w[i]
4  Se s mod 2 ≠ 0 então
5  |  Devolva 0
6  s ← s/2
7  Tomar S[0..n, 0..s] como uma nova matriz
8  Para i ← 0 até n faça
9  |  S[i, 0] ← 1
10 Para j = 0 até s faça
11 |  S[0, j] ← 0
12 Para i = 1 até n faça
13 |  Para j = 0 até s faça
14 |  |  Se w[i] > j então
15 |  |  |  S[i, j] ← S[i − 1, j]
16 |  |  Senão
17 |  |  |  S[i, j] ← max(S[i − 1, j],S[i − 1, j − w[i]])
18 Devolva S[n, W]
```

6.2.3 Problema das Moedas

O Problema das Moedas, também chamado Problema do Troco (PT), é um problema exemplar, o qual podemos modelar uma solução recursiva e aplicar a técnica de programação dinâmica. O problema consiste em descobrir o menor número de moedas (de um país, reino, etc.) que somadas resultam em determinado montante. Este problema surge em situações onde um atendente em uma loja tem que devolver um troco de valor W, desejando somar o menor número de moedas possíveis. O problema das moedas é enunciado em seguida.

Problema das Moedas

Um país tem uma lista de n tipos de moedas com valores $v_1, v_2, \ldots, v_n$ e estoque ilimitado. Dado um inteiro W, o objetivo é encontrar o menor número de moedas que somadas obtenha-se W.

Exemplo. Seja $n = 4$ moedas de valores $\{1, 4, 5, 10\}$ e $W = 7$. O problema das moedas não restringe quanto a quantidade de moedas de um determinado valor que podemos utilizar, desta forma podemos combinar as moedas da seguinte forma

$v_1 \times 7 = 7$
$v_2 \times 1 + v_1 \times 3 = 7$,
$v_3 \times 1 + v_1 \times 2 = 7$,
dentre essas combinações, escolhemos a que utiliza a menor quantidade de moedas. A combinação $1 \times v_3 + 2 \times v_1$ utiliza 3 moedas e é a menor dentre todas. Para obter $W = 8$, somamos duas moedas de valor v_2, $4+4 = 8$. Duas moedas é o menor número de moedas que precisamos somar para obter 8.

É possível chegar em W somando várias combinações do conjunto de moedas.

Exemplo. Considerando o conjunto de quatro moedas com valores $\{1, 5, 10, 20\}$ e montante $W = 20$. Existem 10 maneiras de obtermos 20, especificamente
(1 de v_4), (1 de v_3 + 2 de v_2), (1 de 10 + 1 de 5 + 5 de 1), (1 de 10 + 10 de 1), (2 de 10), (1 de 5 + 15 de 1), (2 de 5 + 10 de 1), (3 de 5 + 5 de 1), (4 de 5), (20 de 1).

A lista acima contém as *soluções viáveis* para este exemplo. Chamamos de **solução viável** uma solução s_i de um conjunto S de soluções possíveis. Assim, $|S|$ é o número de maneiras que podemos obter W e s_i é número total em moedas utilizadas em uma solução possível. Logo, a dificuldade está em encontrar quantas maneiras é possível obter W somando a menor quantidade x_i moedas. Além disso, uma solução minimiza o número de moedas utilizadas.

Podemos definir formalmente o PT como se segue.

Problema das Moedas(PT)	
Instância:	*moedas* $v_1, v_2, \ldots, v_n$;
	objetivo W.
Devolva:	uma n-upla $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$, tal que,
	minimize $P = \sum_{i=1}^{n} x_i$,
	sujeito a $\sum_{i=1}^{n} x_i v_i = W$.

Caracterizando uma solução ótima

Para criar uma solução por programação dinâmica é preciso expressar o problema original em termos de subproblemas. No PT, uma maneira de encolher o problema é considerar todo $w \leq W$, ou ainda, o subconjunto de moedas $v_1, v_2, \ldots, v_i$. Não precisamos manter nenhum tipo de controle em relação a moeda v_i. Sendo assim, definimos a função

$$f(W) = \text{o menor número de moedas para obter W}$$

Em uma solução ótima que dá o menor número de moedas para obter W, existe uma primeira moeda $v_i \leq W$. Se v_i faz parte uma solução ótima, então a solução ótima de um subproblema é definida por $1 + f(W - v_i)$, ou seja, *uma* moeda v_i *mais* $f(W - v_i)$ moedas

para a soma restante $W - v_i$. Não sabemos qual é primeira moeda v_i, por isso calcula-se para todas que não violem a restrição $v_i \leq W$.

Solução recursiva

Novamente, analisando o Problema das Moedas é possível notar que o problema exibe uma estrutura recursiva sobre a qual será definida a equação da programação dinâmica. Ao utilizar a moeda v_i uma instância $W - v_i$ define um subproblema. Ao utilizar a moeda v_{i-1}, de maneira análoga surge o subproblema $W - v_{i-1}$. Denotamos a função f definida em termos de subproblemas

$$f(W) = f(W - v_i), \text{ onde } 1 \leq i \leq n \text{ e } v_i \leq W. \tag{6.5}$$

Para 6.5 ser uma equação de recorrência seu caso base é $f(0) = 0$. No caso em que não existe resposta, denotamos como ∞. Desta forma, temos que

$$f(W) = \begin{cases} \infty & \text{se } W < 0, \\ 0 & \text{se } W = 0, \\ 1 + min(f(W - v_i), 1 \leq i \leq n \text{ e } W \geq v_i) & \text{caso contrário} \end{cases} \tag{6.6}$$

A recursão 6.6 expressa uma solução ótima em termos de soluções ótimas de subproblemas menores.

Calcular uma solução ótima

A partir de 6.6 é definido um algoritmo recursivo que calcula o menor número de moedas que somadas são iguais a W. O algoritmo PTR(W) é definido pela função recursiva em uma abordagem *top-down*.

PTR(W)

Entrada: inteiro W; vetor global v.
Saída: menor número de moedas que somam W.

```
1 Se W = 0 então
2 |  Devolva 0
3 min = ∞
4 Para i ← 1 até n faça
5 |  Se W ≥ v[i] então
6 |  |  temp ← PTR(W − v[i])
7 |  |  Se temp < min então
8 |  |  |  min ← temp
9 Devolva 1+min
```

A complexidade de tempo do algoritmo PTR(W) é exponencial, sendo um algoritmo impraticável para instâncias suficientemente grandes. O problema exibe uma sobreposição de problemas, visto que resolve uma mesma instância de um subproblema repetidas vezes.

Para evitar computar o mesmo subproblema mais de uma vez, aplicamos a estratégia de memoização. Definimos o algoritmo PTM(W) que emprega esta esta ideia. A chamada inicial do algoritmo define um vetor Min[$0, .., n$] com valores -1 para todo M[j] onde $0 \leq j \leq n$.

PTM(W)

```
    Entrada: inteiro W; vetor global v; vetor global Min.
    Saída: menor número de moedas que somam W.
 1  Se Min[W] > −1 então
 2  |  Devolva Min[W]
 3  Se W = 0 então
 4  |  Devolva 0
 5  Min[W] ← ∞
 6  Para i ← 1 até n faça
 7  |  Se W ≥ v[i] então
 8  |  |  Min[W] ← PTM(W − v[i])
 9  |  |  Se temp < Min[W] então
10  |  |  |  Min[W] ← temp
11  Devolva Min[W]+1
```

O próximo passo é elaborar um algoritmo de programação dinâmica em uma *ordem* melhor, iterando desde o caso base até uma solução ótima. O algoritmo PTDP(n, W) resolve o problema por uma abordagem *bottom-up* memoizando solução de subproblemas, sendo o algoritmo PTDP(n, W) uma resposta em programação dinâmica para o PT.

PTDP(n, W)

```
    Entrada: inteiro n, inteiro W; vetor global v; vetor gobal Min.
    Saída: menor número de moedas que somam W.
 1  Tomar Min[0..n] como um novo vetor
 2  Para i ← 0 até n faça
 3  |  Min[i] ← ∞
 4  Min[0] ← 0
 5  Para j ← 1 até n faça
 6  |  Para i ← 1 até W faça
 7  |  |  Se j ≥ M[i] então
 8  |  |  |  temp ← Min[j − v[i]]
 9  |  |  |  Se temp < Min[j] então
10  |  |  |  |  Min[j] ← temp
11  |  |  Min[j] ← 1+Min[j]
12  Devolva Min[W]
```

Analisando a complexidade de tempo do algoritmo PTDP(n, W) temos que o algoritmo

leva $O(nW)$. Assim como no PM, essa complexidade de tempo é dita pseudo-polinomial. A análise de um algoritmo é feita sobre o tamanho sua entrada, como visto na Capítulo 4. A entrada n representa o tamanho de uma sequência de moedas $1, \ldots, n$. O parâmetro W requer $b = \lfloor \log_2 W \rfloor + 1$ bits para ser representado, ou seja, o tamanho de W é dado por 2^b. Portanto, o algoritmo tem complexidade de tempo exponencial $\Theta\left(n2^{\log_2 W}\right)$. O Anexo A.4 disponibiliza uma implementação do algoritmo PTDP(n, W) em linguagem de programação C++.

Problema das Moedas como caso espacial

É possível apresentar uma abordagem para a solução do problema das moedas como um Problema da Mochila. O PM na verdade exibe um *modelo* no qual outros problemas podem se encaixar. Retomando o PM, o problema é definido por um conjunto de n itens de peso $w_1, \ldots, w_n$ e lucro associado $v_1, \ldots, v_n$. Dada a capacidade W da mochila, o objetivo é encontrar um subconjunto de itens que maximiza $\sum_{i=1}^{n} w_i$ dada a condição $\sum_{j=1}^{n} w_j v_j \leq W$.

O PT pode ser modelado como o Problema da Mochila. De acordo com Martello e Toth (1990), o PT é um *caso particular* do Problema da Mochila. Os valores das moedas podem ser modelados como sequência de n valores $v_1, v_2, \ldots, v_n$. Dado uma inteiro W, encontrar uma sequência $w_1, w_2, \ldots, w_n$, onde w_i é interpretado como a frequência com que a moeda v_i é usada, tal que

$$\text{minimize} \sum_{i=1}^{n} w_i$$

sujeito a condição

$$\sum_{i=1}^{n} w_i v_i = W \tag{6.7}$$

condição dada em 6.7 implica que o problema pode não ter resposta.

O primeiro passo para implementar uma solução em programação dinâmica é definir uma função de recorrência. Seja a função

$$MC(i, W) \text{ o menor número de moedas } i \text{ necessárias para soma } W.$$

Utilizando o mesmo argumento para elaborar uma solução em programação dinâmica para o problema PM, uma solução ótima pode ser construída realizando uma sequência de escolhas. A escolha de uma moeda v_i é a decisão de usá-la em uma solução ótima ou não. Para cada moeda existem duas possibilidades de decisão:

Incluir moeda: Tomar a decisão de incluir uma moeda no conjunto de solução ótima resulta em $W - v_i$. Diferente do PM, aqui é possível repetir uma mesma moeda v_i. Logo, deve-se esgotar todas possibilidades M(i,$W - v_i$)utilizando essa moeda respeitando $v_i \leq W$.

Não incluir moeda: Ao ignorar uma moeda, ela não será parte de uma solução ótima, assim resta o subconjunto de moedas $1, 2, \ldots, i-1$ a ser considerado em uma solução ótima M($i-1$,W).

Para que $MC(i, W)$ seja uma função recorrente, é preciso definir o caso base. Não existe resposta se W assumir valor negativo ou se foram testadas todas moedas de valor v_i e ainda não obteve a soma desejada. Nesses dois casos, um valor representativo infinito (∞) é devolvido. A recorrência é dada por

$$MC(i,W) = \begin{cases} \infty & \text{se W<0,} \\ \infty & \text{se i<0 e W>0,} \\ 0 & \text{se W=0,} \\ \min(MC(i-1,W), 1 + MC(i, W - v_i)) & \text{caso contrário} \end{cases} \tag{6.8}$$

Um algoritmo pode ser implementado da equação 6.8. Chamamos de MC(i,W) o algoritmo que devolve o menor número de moedas que somam W. Esse algoritmo tem complexidade exponencial.

MC(i, W)

Entrada: inteiro i; inteiro W; vetor global v.
Saída: menor número de moedas que somam W.

```
1 Se W < 0 então
2 |  Devolva ∞
3 Se i < 0 e W > 0 então
4 |  Devolva ∞
5 Se W = 0 então
6 |  Devolva 0
7 Devolva min(MC(i − 1, W), 1 + MC(i, W − v[i]))
```

As chamadas recursivas do algoritmo resolvem o mesmo problema repetidas vezes, caracterizando uma estrutura de sobreposição, além disso, uma solução ótima e é definida em termo de soluções ótimas de subproblemas. Esses são os requisitos para aplicar uma solução por programação dinâmica. Assim, o algoritmo pode ser melhorado com memoização e uma implementação na ordem *top-down*.

Redefinimos a recorrência 6.8 denotando $M[i, j]$ como a função que calcula o menor número de moedas que somadas são iguais a W. Para todo $1 \leq i \leq n$ e $0 \leq j \leq S$, a função $M[i, j]$ guarda o número de moedas do subproblema i e soma j.

$$M[i,j] = \begin{cases} 0 & \text{se } j = 0 \\ \infty & \text{se } i < 0 \text{ e } j > 0 \\ M[i-1,j] & \text{se } v_i > j \\ \max(M[i-1,j], 1 + M[i-1, j-v_i]) & \text{se } v_i \leq j \end{cases} \tag{6.9}$$

A função 6.9 da origem ao algoritmo de programação dinâmica MCDP(n,W). Com isso, temos uma resposta para o PT modelado através de redução ao Problema da Mochila.

MCDP(n,W)

Entrada: inteiro i; inteiro W; vetor global v.
Saída: menor número de moedas que somam W.

```
1  Tomar M[i..n, j..W] como uma nova matriz
2  Para i ← 0 até n faça
3  |  M[i, 0] ← 0
4  Para j ← 0 até W faça
5  |  M[0, j] ← ∞
6  Para i ← 1 até n faça
7  |  Para j ← 0 até W faça
8  |  |  Se v[i] > j então
9  |  |  |  M[i, j] ← M[i − 1, j]
10 |  |  Senão
11 |  |  |  M[i, j] ← max(M[i − 1, j], 1 + M[i − 1, j − v[i]])
12 Devolva M[n, S]
```

6.2.4 Problema da Subsequência Comum Mais Longa

O problema de processamento de texto, que realiza um teste de similaridade entre duas cadeias de caracteres (*string*), aparece em várias áreas como Genética e Engenharia de Software. Na aplicação genética duas strings podem representar cadeias de DNA as quais gostaríamos de comparar. Em engenharia de software podemos considerar duas versões de um código fonte, de tal forma que, queremos comparar as alterações feitas de uma versão para outra. Determinar a similaridade entre duas strings é algo tão comum que encontramos ferramentas em sistemas operacionais, como o Linux, que disponibilizam um programa para isso, o *diff*. O programa compara duas entradas e diz as diferenças entre cada uma. (GOODRICH; TAMASSIA; MOUNT, 2007)

Dado uma string $X = x_1, x_2, \ldots, x_n$, uma **subsequência** de X é qualquer string $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ onde $i_j < i_{j+1}$, ou seja, uma sequência de caracteres, não necessariamente contínua, mas na mesma ordem que se encontra em X (GOODRICH; TAMASSIA; MOUNT, 2007). A sequência $Z = B, C, D, G$ é uma subsequência de $X = F, A, B, C, B, D, A, B, G$. Dados duas strings X e Y, uma **subsequência comum** Z é uma subsequência de X e Y ao mesmo tempo (CORMEN, 2009). Se $X = A, B, C, G, B, A$ e $Y = F, G, B, A, C, B, A$, a sequência B, C, A é comum a X e Y. A sequência B, C, A não é a subsequência comum *mais longa* (LCS - Longest Common Subsequence, em inglês), pois tem tamanho 3. A subsequência B, C, B, A tem tamanho 4 a qual é comum a X e a Y, portanto uma subsequência comum mais longa.

O problema da LCS é o problema de encontrar a subsequência de tamanho máximo que seja comum as sequências X e Y. Enunciamos o problema da LCS em seguida.

Subsequência Comum Mais Longa

Dado duas cadeias $X = x_1, x_2, \ldots, x_n$ e $Y = y_1, y_2, \ldots, y_n$ encontrar uma subsequência $Z = z_1, z_2, \ldots, z_n$ comum a X e Y ao mesmo tempo com o maior tamanho possível.

Computacionalmente o problema da LCS é definido como se segue.

Subsequência Comum Mais Longa(LCS)	
Instância:	sequências X e Y.
Devolva:	uma subsequência comum mais longa.

Uma solução eficiente para o problema da LCS pode ser descrita utilizando a técnica de programação dinâmica.

Caracterizando uma solução ótima

Em uma solução por força bruta é possível enumerar todas subsequências de X e verificar qual delas também é uma subsequência de Y, mantendo em vista encontrar a de maior tamanho. Gerar cada subsequência de X corresponde a 2^m subsequências, tal que m é o número do índice do último elemento. (CORMEN, 2009).

Seja $X = x_1, x_2, \ldots, x_m$ e $Y = y_1, y_2, \ldots, y_n$ sequências, tomando $Z = z_1, z_2, \ldots, z_k$ como qualquer LCS de X e Y. Na caracterização de uma solução para a LCS existem três condições suficientes para um elemento da sequência X e Y. Essas condições são provadas com mais detalhes em Cormen (2009).

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{K-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_n$ implica que Z é uma LCS de X e Y_{n-1}.

Essas condições definem uma LCS e dizem que uma solução ótima é definida pela sequência comum mais longa dos subproblemas X_{m-1} e Y_{n-1}.

Solução recursiva

Definimos uma função recursiva $L[i, j]$ o tamanho da LCS entre as subsequências X_i e Y_j. Se $i = 0$ ou $j = 0$, uma das duas sequências tem tamanho 0, logo a LCS tem tamanho 0. Da primeira condição, se $x_i = y_j$ temos que a LCS deve ser calculada considerando X_{m-1} e Y_{n-1}. O item dois refere-se ao caso em que $x_m \neq y_n$, então devemos resolver o subproblema de encontrar a LCS de X_{m-1} e Y_n. Por fim, do terceiro item, se $x_m \neq y_n$ calculamos a LCS para X e Y_{m-1}. Essas são as partes da função recursiva $L[i, j]$.

O problema apresenta uma sobreposição de problemas. A solução para o problema calcula a LCS para X e Y_{n-1} e para X_{m-1} e Y. Mas cada um desses subproblemas tem subproblemas ainda menores X_{m-1} e Y_{n-1}.

Definimos a seguinte equação $L[i,j]$ como

$$L[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ L[i-1, j-1] + 1 & \text{se } i,j > 0 \text{ e } x_i = y_j, \\ \max(L[i, j-1], L[i-1, j]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j. \end{cases} \tag{6.10}$$

A equação 6.10 pode ser transformada em algoritmo recursivo de maneira direta. Dado que esse problema exibe a condição de subestrutura ótima e sobreposição de problemas, a técnica de memoização melhora a complexidade de tempo do algoritmo.

Calcular uma solução ótima

Baseados na equação 6.10 $L[i,j]$, é possível escrever um algoritmo recursivo de tempo exponencial que resolva o problema. Uma vez que o LCS tem $\Theta(mn)$ subproblemas distintos, podemos aplicar programação dinâmica para computar as soluções em uma abordagem *bottom-up*. (CORMEN, 2009)

O algoritmo LCS(X,Y), recebe duas strings X e Y como parâmetros. Armazenamos os resultados em uma tabela $n \times m$, o qual é preenchida da esquerda para direita a partir da linha do topo até a última. O algoritmo LCS(X,Y) tem complexidade de tempo $\Theta(nm)$. O Anexo A.5 disponibiliza uma implementação do algoritmo LCS(X, Y) em linguagem de programação C++.

6.2.5 Problema da Maior Subsequência Crescente

O problema da Maior Subsequência Crescente é um problema estudado em ramos da Matemática e Física. Além disso, é problema que tem relação com outros problemas estudados neste trabalho. Enunciamos o problema como se segue.

Maior Subsequência Crescente

Dado uma sequência de inteiros S encontrar a maior subsequência em ordem crescente contida em S.

No problema da *maior subsequência crescente* (LIS – *Longest Increasing Subsequence*, em inglês) sua entrada é uma sequência de números $s_1, s_2, \ldots s_n$ onde s é um elemento da sequência S. Uma *subsequência* é qualquer subconjunto de S tomados em ordem, com a forma $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ onde $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Uma subsequência *crescente* atende a condição $s_{i_j} < s_{i_{j+1}}$ para todo $1 \leq j \leq k$. Nossa tarefa é encontrar a subsequência crescente

LCS(X,Y)

Entrada: sequência X; sequência Y.
Saída: subsequência comum mais longa.

```
1  Tomar m como o tamanho de X
2  Tomar n como o tamanho de Y
3  Tomar L[0...m, 0...n] como uma nova matriz
4  Para i ← 1 até m faça
5  |  L[i, 0] ← 0
6  Para j ← 0 até n faça
7  |  L[0, j] ← 0
8  Para i ← 1 até m faça
9  |  Para j ← 1 até n faça
10 |  |  Se x[i] = y[i] então
11 |  |  |  L[i, j] ← L[i − 1, j − 1] + 1
12 |  |  Senão Se L[i − 1, j] >= L[i, j − 1] então
13 |  |  |  L[i, j] ← L[i − 1, j]
14 |  |  Senão
15 |  |  |  L[i, j] ← L[i, j − 1]
16 Devolva L
```

de maior tamanho entre todas as possíveis (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006). O problema LIS é definido formalmente como se segue.

Maior Subsequência Crescente	
Instância:	sequência S.
Devolva:	maior subsequência crescente.

Exemplo. Uma instância com $S = (6, 3, 5, 2, 7, 8, 1)$. (a) uma subsequência, digamos A_1 pode conter os elementos $5, 3, 1$ da sequência S; (b) A sequência crescente $A_2 = 3, 5, 9$ está contida em S; (c) $A_3 = 2, 7, 8$ é uma subsequência crescente de S.

Para resolver o problema da LIS, uma ideia passa por gerar todos os subsequências de S e verificar qual deles tem é a maior subsequência crescente.

Exemplo. Considere $S = 2, 4, 3, 1$. Podemos gerar todos subconjunto das sequências de S, temos que, $R = \{(2, 4, 3, 1), (2, 4, 3), \ldots, (1)\}$ onde a maior subsequência crescente de S é $(2, 4)$ ou $(2, 3)$. Para esse exemplo, a LIS tem tamanho 2.

Com base na ideia de enumerar todas subsequências e verificar qual o tamanho da maior LIS, um algoritmo de enumeração é descrito como se segue: para cada subsequência R de S, verifique se R é crescente e tem o maior tamanho até aqui e retorne o maior tamanho encontrado. O procedimento EL(S) implementa os passos anteriores.

EL(S)

Entrada: sequência S.
Saída: tamanho da LIS

```
1 max ← 0
2 Para cada subsequência R de S faça
3 |   Se R é crescente e |R| > max então
4 |   |   max ← |R|
5 Devolva max
```

Analisando a complexidade EL(S), são enumerados todos subconjuntos de S para o qual existem 2^n subconjuntos. Para verificar se uma subsequência é crescente, o algoritmo leva $O(n)$. Portanto, EL(S) tem complexidade de $O(n2^n)$. Enumerar todas as possíveis subsequências resolve o problema da LIS, mas é impraticável para quando n cresce.

Em problemas de otimização ou em problemas combinatórios, a programação dinâmica pode ser adequada para implementar uma solução recursiva eficiente (SKIENA, 1998). Aplicando as etapas para elaborar uma solução por programação dinâmica, é preciso caracterizar uma estrutura ótima (etapa 1), definir uma solução recursiva (etapa 2), calcular o valor de uma solução ótima em uma abordagem *bottom-up* (etapa 3).

Caracterizando uma solução ótima

A primeira abordagem de resolução do problema pelo algoritmo EL(S) fornece a ideia do que pode ser a estrutura ótima. Dada uma instância, existe uma subsequência S_i com

termos $s_1, \ldots, s_i$, $s_j < s_i$ que termina no i-ésimo elemento de S. Considerando um elemento s_k, ele pode expandir a subsequência se todo elemento em S_i é menor que s_k, caso contrário, existe S_j, $j < i$, que s_k pode expandir. Definimos a função

$$L[i] \text{ o tamanho da LIS que termina no } i\text{-ésimo elemento} \tag{6.11}$$

Para 6.11 ser uma solução ótima, é precisa saber $L[j]$ para todo $j < i$.

Definindo uma solução recursiva

Para ser uma solução, é preciso escolher o valor de i que maximiza $L[i]$ para todo $i \leq n$ se e somente se $s_j < s_i$, ou seja, para toda subsequência crescente que termina no i-ésimo elemento. Para $L[i]$ ser uma função recursiva, no caso mais básico, $L[1] = 1$. Não sabemos qual i é o maior, assim, calculamos para todas possibilidades. A função 6.11 é definida

$$\begin{aligned} L[1] &= 1 \\ L[i] &= \max_{j<j} (L[j] + 1, \text{ se } s_j < s_i) \end{aligned} \tag{6.12}$$

Calcular o valor de uma solução ótima

O algoritmo equivalente a resolver a função recorrente $L[j]$ é descrito como o algoritmo LR(n,S), onde o parâmetro n é o tamanho da sequência S.

LR(n,S)

Entrada: inteiro n; sequência S.
Saída: tamanho LIS.

```
1 max ← 0
2 Para i ← 1 até n − 1 faça
3 |  Se S[i] < S[n] então
4 |  |  temp ← LR(i, S)
5 |  |  Se temp > max então
6 |  |  |  max ← temp
7 Devolva max + 1
```

Esse algoritmo é uma resposta que resolve o problema da LIS percorrendo toda subsequência de S. Existem no máximo 2^n subsequências, portanto LR(n,S) tem complexidade exponencial $O(2^n)$.

O algoritmo LR(n,S) resolve o mesmo subproblema repetidas vezes. Sendo assim, aplicamos a técnica de memoização de soluções. Reestruturamos o algoritmo para que resolva em uma abordagem *bottom-up*. As alterações no algoritmo concebem ao algoritmo LDP(n,S). Como este algoritmo calcula o valor de cada subproblema apenas uma vez, seu custo de tempo é polinomial de ordem $\Theta(n^2)$. O Anexo A.6 disponibiliza uma implementação do algoritmo LDP(n, S) em linguagem de programação C++.

LDP(n,S)

Entrada: inteiro n; sequência S.
Saída: tamanho da LIS.

```
1  Tomar L[0...n] como um novo vetor
2  Para i ← 1 até n faça
3  |  L[i] ← 1
4  |  Para j ← 1 até i − 1 faça
5  |  |  Se S[i] > S[j] e L[i] < L[j] + 1 então
6  |  |  |  L[i] ← L[j] + 1
7  max ← 0
8  Para i ← 1 até n faça
9  |  Se max < L[i] então
10 |  |  max ← L[i]
11 Devolva max
```

O problema da LIS como um caso especial da LCS

Existe uma relação muito próxima entre o problema da LIS e o problema da LCS. Uma LIS da sequência S é a sequência comum mais longa entre S e T onde T é uma cópia ordena de S. A descrição desse algoritmo é dada pelos passos

1. Faça uma cópia T de S ordenada em $O(n \log n)$.
2. Executar o algoritmo LCS(X,Y) com chamada inicial LCS(S,T)

De fato existem outros algoritmos para LIS, não discutido nesse trabalho. Um deles é implementado como uma busca binária, resultando em uma complexidade tempo $O(n \log n)$. Em Dasgupta, Papadimitriou e Vazirani (2006) o autor reduz o problema da LIS no problema de encontrar o caminho mais longo em grafo direcionado.

6.3 PROVA DE CONCEITO

Os problemas presentes em competições de programação são um excelente exemplo de aplicação e exercício do conceitos apresentados no presente trabalho. Os problemas propostos em competições seguem o formato de **entrada** e **saída** bem definidos. Muitos problemas de competições estão disponíveis na Internet em ambientes chamados *juízes online*.

Juízes online são sistemas nos quais estão reunidos problemas da Computação que abordam os principais temas, como Algoritmos em Grafos, Geometria Computacional, Programação Dinâmica, etc. Estes ambientes são capazes de julgar soluções submetidas (em alguns casos retornando uma porcentagem de acerto) a partir de uma série de testes feitos sobre cada submissão. Juízes como URI Online Judge[1], UVA Online Judge[2] e SPOJ Brasil[3], são exemplos.

[1] ver <http://www.urionlinejudge.com.br>
[2] ver <http://www.uva.onlinejudge.org>
[3] ver <http://br.spoj.com>

Tais ambientes são úteis a competidores e interessados pois se assemelham aos ambientes de competição.

6.3.1 Problema do Banco Inteligente

O problema "O Banco Inteligente" é um problema da OBI (Olimpíada Brasileira de Informática)[4]. Este problema pode ser resolvido utilizando um algoritmo de programação dinâmica.

O problema consiste em um "banco inteligente" que dá aos clientes a possibilidade de escolher as notas no momento de um saque de um valor S. O problema disponibiliza notas de 2, 5, 10, 20, 50 e 100. Computacionalmente o problema é descrito como se segue.

O Banco Inteligente
Entrada: A primeira linha da entrada contém um inteiro S, o valor do saque. A segunda linha contém seis inteiros $N_2, N_5, N_{10}, N_{20}, N_{50}$ e N_{100}, respectivamente, o número de notas de valores 2,5,10,20,50 e 100, disponíveis na máquina.
Saída: Um inteiro, o número de formas distintas da máquina entregar o saque.

Fazendo alterações no algoritmo MTDP(n,W) da Seção 6.2.3 é possível implementar um algoritmo que resolva este problema de maneira eficiente.

6.3.2 Problema K

O Problema K (Trabalho do Papa) da XV Maratona de Programação[5], competição interna da Universidade de São Paulo é um problema no qual a utilização da técnica de programação dinâmica pode ser utilizada com sucesso.

Em uma fábrica de produtos químicos, existe a tarefa de acomodar pacotes de produtos em pilhas. O problema surge do fato de que alguns destes pacotes eram muito pesados e outros tinham pouca resistência a ter peso sobre eles. Contudo, os gerentes sempre querem pilhas com o maior tamanho possível. A tarefa consiste em determinar o número máximo de pacotes que podem ser empilhados sem ultrapassar a resistência de nenhuma caixa da pilha. O problema computacional é descrito em seguida.

A resposta para esse problema é dada por um algoritmo muito semelhante ao algoritmo do Problema da Mochila, discutido na Seção 6.2.1. O detalhe é ficar atento o que compõe os estados da árvore de busca.

[4] ver <http://olimpiada.ic.unicamp.br/pratique/programacao/nivel2/2015f1p2_banco>
[5] ver <https://www.ime.usp.br/~cef/XVmaratona/>

Problema K	
Entrada:	A entrada é composta por diversas linhas. A primeira linha contém um inteiro N, indicando o número de caixas. Cada uma das N linhas seguintes contém inteiros P e R, correspondendo, respectivamente, ao peso e resistência de uma caixa.
Saída:	O maior número de caixas que podem ser empilhadas nas condições do enunciado..

7 CONCLUSÃO

Problemas computacionais podem surgir em diferentes áreas e contextos, iniciando uma demanda por habilidades voltadas às técnicas de Projeto de Algoritmos. Programadores que buscam se diferenciar em suas carreiras devem dominar o maior número de técnicas possíveis. Assim, torna-se relevante um amplo repertório nesse sentido. As técnicas de *Backtracking* e Programação Dinâmica são possibilidades, pois ajudam na resposta de um problema computacional de forma concisa e eficiente – uma solução bem construída pode economizar recursos, sejam eles computacionais (tempo e espaço, por exemplo) ou não.

A técnica de projeto *Backtracking* é uma das técnicas frequentes na construção de soluções combinatórias onde o objetivo é uma solução ótima. Se difere da técnica *Branch and Bound*, uma vez que esta última, tenta decidir se um estado é uma solução viável rapidamente, sem a necessidade de percorrer todo espaço de busca. A relação de algoritmos *backtracking* com a busca em profundidade torna a técnica poderosa em problemas cujo o espaço de busca é implícito.

Soluções em Programação Dinâmica podem surgir a partir de relações de recorrência que revelam a característica de subestrutura ótima e sobreposição de subproblemas, sendo uma técnica poderosa com aplicações em diversos problemas. Algumas linguagens de programação, como a linguagem LISP, utilizam a ideia de memoização de respostas.

A menos que $\mathcal{P} = \mathcal{NP}$, para os problemas apresentados no Capítulos 5 e no Capítulo 6, não existem soluções de tempo polinomial. Uma das abordagens de maior sucesso no desenvolvimento de algoritmos para resolução do problema da Clique Máxima tem sido a técnica de *Branch and Bound*. Seguindo esse caminho, a solução do problema da Clique Máxima pelo algoritmo MCBB é uma forma de implementar de maneira unificada e natural vários algoritmos. Essa abordagem pode ser útil em experimentos cujo objetivo é comparar o desempenho entre vários algoritmos.

A solução do Problema da Mochila por programação dinâmica se mostra viável do ponto de vista de implementação e também do ponto de vista de complexidade de tempo. Além dessa abordagem de resolução, o problema pode ser resolvido por Algoritmos de Aproximação, uma técnica não abordada por este trabalho.

O ensino de Computação tem se tornado uma habilidade importante nas mais variadas áreas. Campos como os da Ciência, Indústria, Comércio e Educação (formal ou não-formal) estão cada vez mais imersos em recursos computacionais. Possuir as habilidades necessárias para construção de conceitos e interação com esses meios tem sido uma tendência, a qual tende a continuar crescendo frente aos avanços e barreiras contemporâneos. Sendo assim, as técnicas presentes nesse trabalho também tem o poder de corroborar nos aspectos educacionais de Computação.

Como trabalhos futuros, é possível produzir recursos educacionais abertos a respeito

dos conceitos e das técnicas apresentadas neste trabalho. A Wikitona é um repositório *online* que ainda não cobre estes assuntos. De fato, a Wikitona é um repositório aberto voltado para interessados em competições de programação. Por serem assuntos recorrentes em competições, torna-se relevante produzir uma contribuição para competidores e interessados. Além disso, experimentos com algoritmos implementados em diferentes linguagens, resolvendo problemas $\mathcal{NP}$-difíceis, são oportunidades de estudos em trabalho posteriores.

REFERÊNCIAS

AHO, A. V.; HOPCROFT, J. E. **The design and analysis of computer algorithms**. [S.l.]: Pearson Education India, 1974. Citado na página 32.

AKKOYUNLU, E. The enumeration of maximal cliques of large graphs. **SIAM Journal on Computing**, SIAM, v. 2, n. 1, p. 1–6, 1973. Citado na página 42.

ANDREI, A.; MAHAVIER, W. T. Teaching the backtracking method using intelligent games. **ACET Journal of Computer Education & Research**, 2014. Citado 2 vezes nas páginas 14 e 16.

BARCELOS, T. S.; SILVEIRA, I. F. Pensamento computacional e educação matemática: Relações para o ensino de computação na educação básica. In: SBC. **XX Workshop sobre Educação em Computação, Curitiba. Anais do XXXII CSBC**. Curitiba, 2012. v. 2, p. 23. Citado na página 15.

BARUAH, S.; FISHER, N. The partitioned multiprocessor scheduling of sporadic task systems. In: IEEE. **Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International**. [S.l.], 2005. p. 9–pp. Citado na página 61.

BELLMAN, R. On the theory of dynamic programming. **Proceedings of the National Academy of Sciences**, National Acad Sciences, v. 38, n. 8, p. 716–719, 1952. Citado na página 53.

BELLMAN, R. **The theory of dynamic programming**. [S.l.], 1954. Citado na página 53.

BELLMAN, R. **Dynamic programming**. [S.l.]: Courier Corporation, 2013. Citado na página 53.

BOMZE, I. M. et al. The maximum clique problem. In: **Handbook of combinatorial optimization**. [S.l.]: Springer, 1999. p. 1–74. Citado 4 vezes nas páginas 13, 28, 41 e 42.

BONDY, J. A.; MURTY, U. S. R. **Graph theory with applications**. London: Macmillan London, 1976. v. 290. Citado 2 vezes nas páginas 13 e 23.

BRASIL, Ministério da Educação. **Orientações Curriculares para ensino médio**. Brasília: MEC/SEB, 2002. v. 2. Citado na página 14.

BRON, C.; KERBOSCH, J. Algorithm 457: finding all cliques of an undirected graph. **Communications of the ACM**, ACM, v. 16, n. 9, p. 575–577, 1973. Citado na página 42.

CARMO, R.; ZÜGE, A. Branch and bound algorithms for the maximum clique problem under a unified framework. **Journal of the Brazilian Computer Society**, Springer, v. 18, n. 2, p. 137–151, 2012. Citado 2 vezes nas páginas 44 e 45.

CAZALS, F.; KARANDE, C. A note on the problem of reporting maximal cliques. **Theoretical Computer Science**, Elsevier, v. 407, n. 1-3, p. 564–568, 2008. Citado na página 44.

CORMEN, T. H. **Introduction to algorithms**. Cambridge, Massachusetts: MIT press, 2009. Citado 7 vezes nas páginas 13, 51, 53, 54, 69, 70 e 71.

CORREA, M. V.; ALBUQUERQUE, J.; ZÜGE, A. P. Wikitona: Um repositório de algoritmos para competições de programação. **Simpósio de Licenciaturas em Ciências Exatas e em Computação (SLEC)**, 2016. Citado 2 vezes nas páginas 14 e 18.

DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. **Algorithms**. [S.l.]: McGraw-Hill, Inc., 2006. Citado 5 vezes nas páginas 13, 14, 53, 72 e 75.

DREYFUS, S. Richard bellman on the birth of dynamic programming. **Operations Research**, Informs, v. 50, n. 1, p. 48–51, 2002. Citado na página 53.

DUARTE, E. P. et al. Finding stable cliques of planetlab nodes. In: IEEE. **Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on**. [S.l.], 2010. p. 317–322. Citado na página 41.

ERDŐSNÉ NÉMETH, Á.; ZSAKÓ, L. The place of the dynamic programming concept in the progression of contestants' thinking. **Olympiads in Informatics**, 2016. Citado 2 vezes nas páginas 14 e 17.

FORIŠEK, M. Towards a better way to teach dynamic programming. **Olympiads in Informatics**, p. 45, 2015. Citado 2 vezes nas páginas 14 e 17.

FORIŠEK, M.; STEINOVÁ, M. Metaphors and analogies for teaching algorithms. In: ACM. **Proceedings of the 43rd ACM technical symposium on Computer Science Education**. [S.l.], 2012. p. 15–20. Citado 2 vezes nas páginas 14 e 18.

GAREY, M. R.; JOHNSON, D. S. **Computers and intractability**. San Francisco: Freeman New York, 2002. v. 29. Citado na página 13.

GERSTING, J. L. **Fundamentos matemáticos para a ciência da computação: um tratamento moderno de matemática discreta**. Rio de Janeiro: Livros Técnicos e Científicos, 2004. Citado na página 19.

GOODRICH, M.; TAMASSIA, R.; MOUNT, D. **DATA STRUCTURES AND ALGORITHMS IN C++**. [S.l.]: John Wiley & Sons, 2007. Citado 2 vezes nas páginas 14 e 69.

GROVER, S.; PEA, R. Computational thinking in k–12: A review of the state of the field. **Educational Researcher**, SAGE Publications Sage CA: Los Angeles, CA, v. 42, n. 1, p. 38–43, 2013. Citado 2 vezes nas páginas 15 e 16.

HALIM, S. et al. **Competitive Programming 3**. [S.l.]: Lulu Independent Publish, 2013. Citado na página 10.

HARARY, F.; ROSS, I. C. A procedure for clique detection using the group matrix. **Sociometry**, JSTOR, v. 20, n. 3, p. 205–215, 1957. Citado na página 42.

HOROWITZ, E.; SAHNI, S. Computing partitions with applications to the knapsack problem. **Journal of the ACM (JACM)**, ACM, v. 21, n. 2, p. 277–292, 1974. Citado na página 62.

HOROWITZ, E.; SAHNI, S. **Fundamentals of computer algorithms**. New York, NY: Computer Science Press, 1978. Citado 2 vezes nas páginas 34 e 38.

KARMARKAR, N.; KARP, R. M. **The Differencing Method of Set Partitioning**. [S.l.], 1982. Citado na página 62.

KARP, R. M. Reducibility among combinatorial problems. In: **Complexity of computer computations**. [S.l.]: Springer, 1972. p. 85–103. Citado 3 vezes nas páginas 41, 45 e 61.

KELLERER, H. et al. Semi on-line algorithms for the partition problem. **Operations Research Letters**, Elsevier, v. 21, n. 5, p. 235–242, 1997. Citado na página 61.

KELLERER, H.; PFERSCHY, U.; PISINGER, D. **Knapsack problems**. New York: Springer, Berlin, 2003. Citado 5 vezes nas páginas 24, 25, 45, 48 e 54.

KLEINBERG, J.; TARDOS, E. **Algorithm design**. [S.l.]: Pearson Education India, 2006. Citado 2 vezes nas páginas 11 e 35.

KNUTH, D. E. **The art of computer programming: sorting and searching**. [S.l.]: Pearson Education, 1998. v. 3. Citado na página 34.

KOCH, I. Enumerating all connected maximal common subgraphs in two graphs. **Theoretical Computer Science**, Elsevier, v. 250, n. 1-2, p. 1–30, 2001. Citado na página 44.

KORF, R. E. A complete anytime algorithm for number partitioning. **Artificial Intelligence**, Elsevier, v. 106, n. 2, p. 181–203, 1998. Citado 2 vezes nas páginas 14 e 62.

KOZEN, D. C. **The design and analysis of algorithms**. College Park, Maryland: Springer Science & Business Media, 2012. Citado na página 59.

KREHER, D. L.; STINSON, D. R. **Combinatorial algorithms: generation, enumeration, and search**. Boca Raton, Florida: CRC Press, 1999. (Discrete Mathematics and Its Applications). ISBN 978-0849339882. Citado 8 vezes nas páginas 13, 28, 29, 30, 31, 36, 38 e 47.

LEVITIN, A.; MUKHERJEE, S. **Introduction to the design & analysis of algorithms**. [S.l.]: Pearson Education, 2011. v. 3. Citado 2 vezes nas páginas 14 e 32.

MAGAZINE, M. J.; NEMHAUSER, G. L.; JR, L. E. T. When the greedy solution solves a class of knapsack problems. **Operations Research**, INFORMS, v. 23, n. 2, p. 207–217, 1975. Citado na página 48.

MARTELLO, S.; TOTH, P. **Knapsack Problems: Algorithms and Computer Implementations**. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN 0-471-92420-2. Citado 5 vezes nas páginas 14, 48, 60, 61 e 67.

PPC, Projeto Pedagódico do Curso de Licenciatura em Computação – Versão 2.0. **PPC, Projeto Pedagódico do Curso de Licenciatura em Computação – Versão 2.0**. Jandaia do Sul: Curso de Licenciatura em Computação, Universidade Federal do Paraná – Campus Jandaia do Sul, 2015. Citado na página 16.

SEDGEWICK, R. **Algorithms in C**. Massachusetts: Addison-Wesley, Reading, Massachusetts, 1990. Citado 2 vezes nas páginas 13 e 49.

SEDGEWICK, R.; FLAJOLET, P. **An introduction to the analysis of algorithms**. [S.l.]: Pearson Education India, 1996. Citado na página 35.

SHAMIR, A. A polynomial-time algorithm for breaking the basic merkle-hellman cryptosystem. **IEEE transactions on information theory**, IEEE, v. 30, n. 5, p. 699–704, 1984. Citado 2 vezes nas páginas 24 e 46.

SKIENA, S. S. **The algorithm design manual**. [S.l.]: Springer Science & Business Media, 1998. v. 1. Citado 6 vezes nas páginas 11, 32, 33, 49, 52 e 73.

SKIENA, S. S.; REVILLA, M. A. **Programming challenges: The programming contest training manual**. New York: Springer Science & Business Media, 2006. Citado 5 vezes nas páginas 10, 13, 36, 37 e 40.

TOMITA, E.; KAMEDA, T. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. **Journal of Global Optimization**, Springer, v. 37, n. 1, p. 95–111, 2007. Citado na página 44.

TOMITA, E. et al. A simple and faster branch-and-bound algorithm for finding a maximum clique. In: SPRINGER. **International Workshop on Algorithms and Computation**. Berlin, Heidelberg, 2010. p. 191–203. Citado na página 44.

TOMITA, E.; TANAKA, A.; TAKAHASHI, H. The worst-case time complexity for generating all maximal cliques and computational experiments. **Theoretical Computer Science**, Elsevier, v. 363, n. 1, p. 28–42, 2006. Citado na página 44.

UNESCO. **Declaração REA de Paris em 2012**. 2012. <http://www.unesco.org/fileadmin/MULTIMEDIA/HQ/CI/WPFD2009/Portuguese_Declaration.html>. (Accessed on 04/22/2017). Citado 2 vezes nas páginas 14 e 18.

WING, J. M. Computational thinking. **Communications of the ACM**, v. 49, n. 3, p. 33–35, 2006. Citado na página 15.

ZÜGE, A. P. **Solução exata do problema da clique máxima**. Dissertação (Mestrado) — Universidade Federal do Paraná, nov. 2011. Disponível em: <http://dspace.c3sl.ufpr.br/dspace/handle/1884/27588>. Citado na página 44.

APÊNDICES

APÊNDICE A – IMPLEMENTAÇÕES

Em seguida são apresentados algoritmos, descritos no decorrer do presente trabalho, implementados em linguagem de programação C++ utilizando a *Standard Template Library* (STL).

A.1 ALGORITMO BACKTRACKING PARA O PROBLEMA DAS RAINHAS

Implementação em linguagem de programação C++ do algoritmos escrito na Seção 5.2 utilizando a STL.

Implementação A.1 – Algoritmo para o Problema das Rainhas

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <vector>
4  using namespace std;
5  #define MAX 100
6  int a[MAX], n, sol;
7
8  vector<int> places(int k) {
9      vector<int> C;
10     bool safe;
11     for (int i = 1; i <= n; i++) {
12         safe = true;
13         for (int j = 1; j < k; j++) {
14             if (abs(k-j) == abs(a[j]-i))
15                 safe = false;
16             if (a[j] == i)
17                 safe = false;
18         }
19         if (safe == true)
20             C.push_back(i);
21     }
22     return C;
23 }
24 void rainhas(int k) {
25     if (k == n)
26         sol++;
27     else {
```

```
28          k++;
29          vector<int> C = places(k);
30          for (auto c: C) {
31              a[k] = c;
32              rainhas(k);
33          }
34      }
35  }
36
37  int main() {
38      sol = 0, n=6;
39      rainhas(0);
40      printf("%d\n",sol);
41      return 0;
42  }
```

A.2 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA O PROBLEMA DA MOCHILA

Implementação em linguagem de programação C++ do algoritmo MDP(n, W) escrito na Seção 6.2.1 utilizando a STL.

Implementação A.2 – Algoritmo para o Problema da Mochila

```
1  #include <cstdio>
2  #include <algorithm>
3
4  using namespace std;
5  #define MAX 1000
6
7  int w[MAX];
8  int v[MAX];
9
10 int Knapsack(int n, int W) {
11     int M[n+1][W+1];
12     for (int i = 0; i <= n; i++)
13         M[i][0] = 0;
14     for (int j = 0; j <= W; j++)
15         M[0][j] = 0;
16     for (int i = 1; i <= n; i++) {
17         for (int j = 0; j <= W; j++) {
18             if (w[i-1] > j)
```

```
19                  M[i][j] = M[i-1][j];
20              else
21                  M[i][j] = max(M[i-1][j], v[i-1] + M[i-1][j - w[i-1]]);
22          }
23      }
24      return M[n][W];
25  }
26
27  int main() {
28      int W = 11; int n=4;
29      w[0]=7, w[1]=4, w[2]=5, w[3]=3;
30      v[0]=31, v[1]=15,v[2]=17,v[3]=10;
31      printf("%d\n", Knapsack(n,W));
32      return 0;
33  }
```

A.3 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA O PROBLEMA DA PARTIÇÃO

Implementação em linguagem de programação C++ do algoritmo Partition(n, W) escrito na Seção 6.2.2 utilizando a STL.

Implementação A.3 – Agloritmo para o Problema da Partição

```
1  #include <cstdio>
2  #include <algorithm>
3  using namespace std;
4
5  bool partition(int n, int v[]) {
6      int s = 0;
7
8      for (int i = 0; i < n; i++)
9          s += v[i];
10
11     if (s%2 != 0)
12         return false;
13
14     s /= 2;
15     bool M[n+1][s+1];
16     for (int i = 0; i <= n; i++)
17         M[i][0] = true;
18     for (int j = 1; j <= s; j++)
```

```
19          M[0][j] = false;
20      for (int i = 1; i <= n; i++) {
21          for (int j = 1; j <= s; j++) {
22              if (j < v[i-1])
23                  M[i][j] = M[i-1][j];
24              else
25                  M[i][j] = M[i-1][j] || M[i-1][j-v[i-1]];
26          }
27      }
28      return M[n][s];
29  }
30
31  int main() {
32      int n=6;
33      int v[] = {2,7,7,12,5,3};
34      if (partition(n,v))
35          printf("Y\n");
36      else
37          printf("N\n");
38  }
```

A.4 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA O PROBLEMA DAS MOEDAS

Implementação em linguagem de programação C++ do algoritmo PTDP(n, W) escrito na Seção 6.2.3 utilizando a STL.

Implementação A.4 – Algoritmo para o Problema das Moedas

```
1  #include <cstdio>
2  #include <vector>
3  #include <cstring>
4
5  using namespace std;
6  #define INF 0x3fffffff
7
8  int dp(int m[], int x, int n) {
9      vector<int> f(x+1,INF);
10     f[0] = 0;
11     for (int i = 1; i <= x; i++) {
12         for (int j = 0; j < n; j++) {
13             if (i >= m[j]) {
```

```
14                  int t = f[i - m[j]];
15                  if (t < f[i])
16                      f[i] = t;
17              }
18          }
19          f[i] = f[i]+1;
20      }
21      return f[x];
22  }
23  int main() {
24      int m[] = {2,5,10,20}, x, n;
25      n = 4; x = 256;
26      printf("%d\n", dp(m,x,n));
27      return 0;
28  }
```

A.5 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA O PROBLEMA DA SUBSEQUÊNCIA COMUM MAIS LONGA

Implementação em linguagem de programação C++ do algoritmo LCS(X, Y) escrito na Seção 6.2.4 utilizando a STL.

Implementação A.5 – Algoritmo para o Problema da LCS

```
1  #include <cstdio>
2  #include <algorithm>
3  #include <string>
4  using namespace std;
5
6  int lcs(char *X, char *Y, int m, int n) {
7      int L[m+1][n+1];
8      for (int i = 0; i <= m; i++) {
9          for (int j = 0; j <= n; j++) {
10             if (i == 0 || j == 0)
11                 L[i][j] = 0;
12             else if (X[i-1] == Y[i-1])
13                 L[i][j] = L[i-1][j-1] + 1;
14             else
15                 L[i][j] = max(L[i-1][j], L[i][j-1]);
16         }
17     }
```

```
18      return L[m][n];
19  }
20
21  int main() {
22      char X[] = "abcbdab";
23      char Y[] = "bdcaba";
24      int n, m;
25      n = 7, m = 6;
26      printf("%d\n", lcs(X,Y,n,m));
27      return 0;
28  }
```

A.6 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA O PROBLEMA DA MAIOR SUBSEQUÊNCIA CRESCENTE

Implementação em linguagem de programação C++ do algoritmo LIS(n, S) escrito na Seção 6.2.5 utilizando a STL.

Implementação A.6 – Algoritmo para o Problema da LIS

```
1  #include <cstdio>
2
3  #define MAX 100
4
5  int dp(int S[], int N) {
6      int maxLength = 1, L[MAX];
7      for (int i = 0; i < N; i++) {
8          L[i] = 1;
9          for (int j = 0; j <= i-1; j++) {
10             if (S[i] > S[j] && L[i] < L[j]+1) {
11                 L[i] = L[j] + 1;
12             }
13         }
14         if (L[i] > maxLength) {
15             maxLength = L[i];
16         }
17     }
18     return maxLength;
19 }
20 int main() {
21     int S[] = {5,2,8,6,3,6,9,7};
```

```
22      int n = 8;
23      printf("%d\n", dp(S,n));
24      return 0;
25  }
```